The Pragmatic Programmers

TURING 图灵程序设计丛书

构建高性能JVM应用、将开发效率提高几个数量级，
从掌握Groovy开始。

# Groovy程序设计

【美】Venkat Subramaniam 著
臧秀涛 译



人民邮电出版社
POSTS & TELECOM PRESS

# 数字版权声明

## 作者介绍

### Venkat Subramaniam

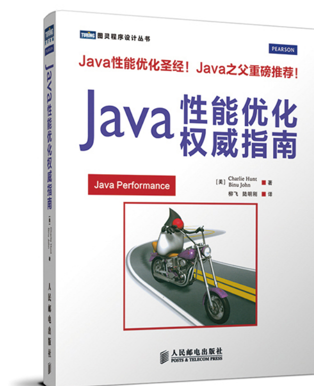
Agile Developer公司创始人，敏捷开发权威人士。他培训并指导了美国、加拿大、印度和欧洲多国的上千名软件开发人员，并多次在各种大会上发表演讲。除本书外，还著有Jolt大奖图书《高效程序员的45个习惯：敏捷开发修炼之道》。

## 译者介绍

### 臧秀涛

硕士毕业于中国科学院计算技术研究所。曾从事网络游戏、操作系统等方面的开发工作。喜爱编程语言和编译器相关技术。热爱读书和翻译。

## 推荐阅读

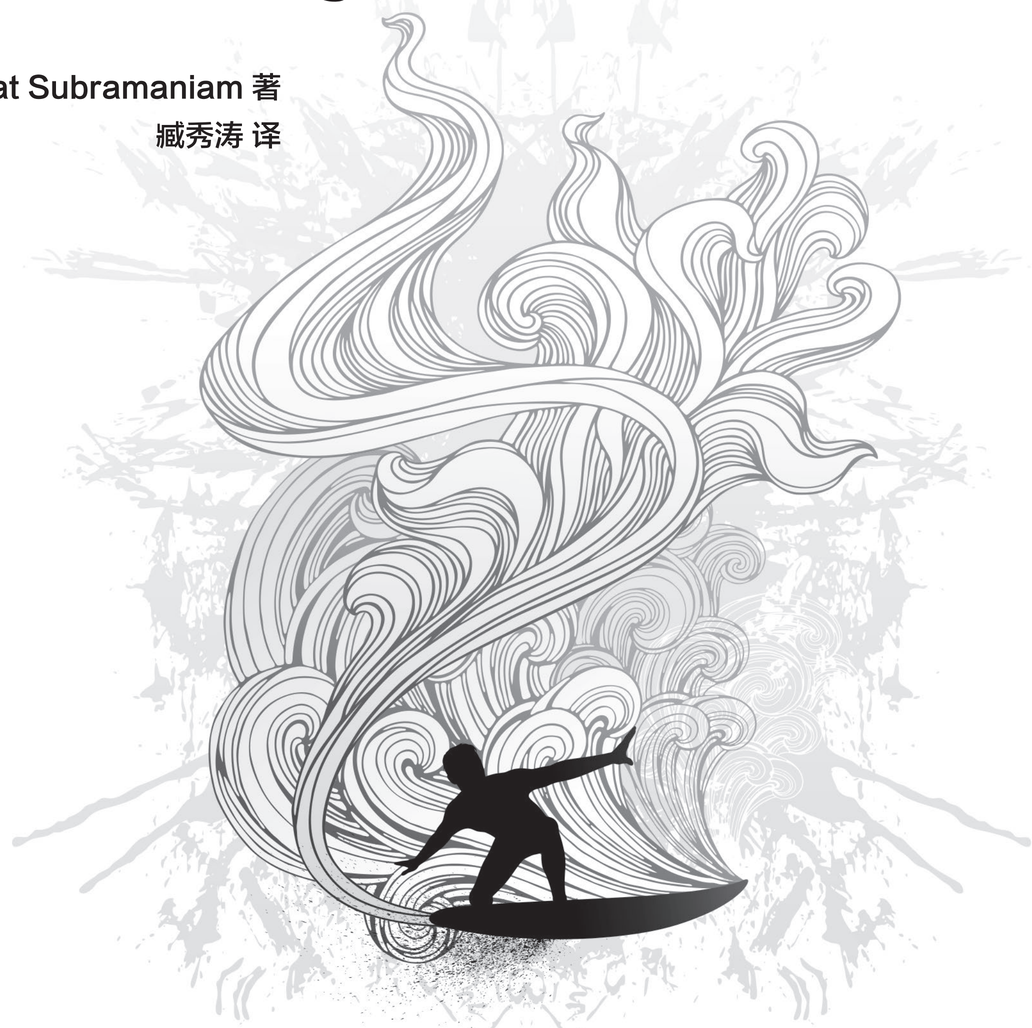


TURING 图灵程序设计丛书

# Groovy程序设计


【美】Venkat Subramaniam 著

臧秀涛 译


人 民 邮 电 出 版 社

北　京

图书在版编目（CIP）数据

Groovy程序设计 / (美) 苏帕拉马尼亚姆
(Subramaniam,V.) 著 ; 臧秀涛译. -- 北京 : 人民邮电
出版社, 2014.11
（图灵程序设计丛书）
ISBN 978-7-115-37041-9

Ⅰ. ①G… Ⅱ. ①苏… ②臧… Ⅲ. ①程序语言－程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第209039号

## 内 容 提 要

本书是 Groovy 编程指南，结合诸多实例探索了 Groovy 语言特性。主要内容包括：Groovy 基础知识介绍、如何将 Groovy 应用于日常编码、MOP 与元编程、使用元编程等。

本书适合 Java 开发人员学习 Groovy，对 Groovy 已有了解的程序员也可在本书中学到一些其他书中无从提及的诀窍和技巧。

◆ 著　　[美] Venkat Subramaniam
译　　臧秀涛
责任编辑　朱　巍
执行编辑　裴　阳
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：18.5
字数：437千字　　　　2014年 11 月第 1 版
印数：1 – 3 000册　　　　2014年 11 月北京第 1 次印刷

著作权合同登记号　图字：01-2013-8466号

定价：69.00元

**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**

**反盗版热线：(010)81055315**

**广告经营许可证：京崇工商广字第 0021 号**

# 版权声明

# 序

Venkat 曾著书引导读者学习 Groovy 1.5 的所有功能特性，助其成为娴熟的 Groovy 开发者。俗话说，光阴似箭。现在是时候探索一下 Groovy 2 都有哪些功能特性了。当然，Venkat 这位深受读者喜爱的作家都为我们考虑到了。

对于Groovy的2.0版本，我们Groovy团队主要把精力投放在了以下三个方面。首先，使Groovy与 JDK 7 接轨：添加了 Java 7 “Project Coin”所带来的语法增强；用 `invokedynamic` 字节码指令和内部的 API 来支撑 Groovy 的运行时。这样一来，即使用的是比较老的 JDK，也可以使用最新添加的语法。当然，如果运行 JDK 7 的话，还可以获得更好的性能体验。

其次，我们将 Groovy 分解成较小型的模块，包括一个核心模块和一些 API 相关的模块，所以你可以选择感兴趣的部分来组织自己的应用。我们还扩展了 Groovy 开发包（Groovy Development Kit），支持开发者创建自己的扩展方法，就像 Groovy 用著名的 `DefaultGroovyMethods` 类对 JDK 所做的增强那样。

最后，还有一点同样重要，我们引入了一个“静态”（static）主题，它包括两个比较新奇的地方：静态类型检查和静态编译。借助前者，我们可以在编译时轻松地捕获输入拼写错误及其他错误，甚至还支持对领域特定语言（Domain-Specific Language）进行类型检查；借助后者，对于应用中要求最高性能的关键部分，我们可以获得与 Java 同样的性能。

有了这些对语言和 API 的增强，Groovy 如美酒佳酿般继续趋向成熟；而 Venkat 就像乐于分享专长的调酒师，将他所知道的 Groovy 的所有强大特性，通过我们正要阅读的这本结构合理的书分享出来，帮助读者紧跟语言发展的步伐，同时更上一层楼。

Guillaume Laforge<br>
Groovy 项目管理者<br>
2013 年 6 月

# 引　言

Java平台可以说是当下功能最为强大、应用最为广泛的生态系统之一。它有3个重要的组成部分。

- Java 虚拟机（Java Virtual Machine，JVM）。这些年来，JVM 已经变得越来越强大，性能也越来越好。
- Java 开发包（Java Development Kit，JDK）。包括丰富的第三方类库和框架，可以帮助我们有效地利用 Java 平台。
- 基于 JVM 的语言集合。Java 语言当然是第一位的，这些语言集合可以帮助我们在 Java 平台上编写程序。

语言就像能使我们在平台上航行的交通工具，通过这些交通工具我们可以轻松抵达该平台的各个部分。截至目前，Java语言已经有了长足的进步，其类库也被重构和扩充过。尽管Java语言带给我们诸多好处，我们还是需要超越Java，寻找更为轻量级且高效的语言。如果使用得当，动态语言、函数式编程风格和元编程功能可以帮助我们更快速地航行。还以交通工具作比，这些可不是更快的汽车，而是飞行器，一种能将开发效率提高几个数量级的飞行器。

Java语言一直想引入元编程和函数式编程风格，但却总是摇摆不定。未来的版本将对其中的某些特性提供不同程度的支持[①]。然而，我们不必等到那一天。现在，就在此时此刻，使用Groovy就可以利用所有的动态功能构建高性能的JVM应用。

## Groovy是什么

韦氏词典对groovy一词的定义是marvelous、wonderful、excellent、hip、trendy，有“非凡、绝妙、优秀和时髦”等意义。Groovy语言集合了上面这一切优点，它是轻量级的，限制较少，而且还是动态、面向对象的，并且运行在JVM上。Groovy基于Apache 2.0许可协议开源。它博采诸如Smalltalk、Python和Ruby等众家语言之长，同时保留了Java程序员熟悉的语法。Groovy编译为Java字节码，它还扩充了Java API和类库。Groovy基于Java 1.5及更高版本运行。要部署的话，除了常规的Java及其组件外，我们需要的就是一个Groovy的JAR文件，而Java的东西我们都已准备好了。

① Java 8已于2014年3月发布，带来了Lambda表达式，支持一定程度的函数式编程。——译者注

Groovy是一门“几经重生”的语言。[1] 该语言由James Strachan和Bob McWhirter于2003年启动开发，之后于2004年3月成为JSR 241（Java Specification Request，JSR，即Java规范请求）。不久，因为存在一些困难和问题，该语言几近被放弃。Guillaume Laforge和Jeremy Rayner决定再次努力，并使Groovy重获新生。他们首先修复了一些bug，同时将语言特性稳定下来。前途未卜这种状态持续了一段时间，包括提交者和用户在内的很多人干脆放弃。最后，一群聪明热情的开发者加入到Guillaume和Jeremy的行列，一个充满生气的开发者社区形成了。

Groovy的1.0版本发布于2007年1月2日。令人鼓舞的是，在1.0版本发布之前，美国和欧洲的很多组织已经将Groovy应用于商业项目。一些组织和开发者开始在其项目的各个层次上使用Groovy，在产业中大量应用该语言的时机已然成熟。2012年的年中，Groovy的2.0版本发布了。

像Grails、CodeNarc、easyb、Gradle和Spock这样的框架和工具都是Groovy的闪光点。其中，Grails[2]是一种基于“规约编程”（coding by convention）的动态Web开发框架，用到了Groovy的元编程功能。通过Grails，我们可以使用Groovy、Spring、Hibernate以及其他Java框架快速构建JVM上的Web应用。

## 为何要使用动态语言

动态语言能够在运行时扩展程序，包括修改类型、行为和对象结构。利用动态语言，静态语言在编译时做的一些事情，我们可以在运行时做，甚至可以执行在运行时即时创建的程序语句。

例如，要计算8万美元的薪水提高5%是多少，只要这样写就行了：

```
5.percentRaise(80000)
```

没错，这就是`java.lang.Integer`对动态方法的友好响应。动态方法很容易添加，像这样即可：

```
Integer.metaClass.percentRaise = { amount -> amount * (1 + delegate / 100.0) }
```

可见，在Groovy中向类中添加动态方法非常容易。我们向使用`delegate`变量引用的`Integer`实例中添加了一个动态方法，负责返回增加相应百分比之后的美元数。

动态语言的灵活性给我们带来了在应用执行时演进程序的优势。这远远超越了代码生成。代码生成是20世纪才会考虑的技术。实际上，生成的代码就像持续的瘙痒，如果一直挠，就会转变为伤痛。而动态语言有更好的方式。代码合成（code synthesis）是运行时在内存中创建代码，动态语言使得代码合成更容易被人接受了。代码基于应用的逻辑流程合成，并即时（just in time）

① 参见Guillaume Laforge的博客文章“A bit of Groovy history”（Groovy的一点历史）http://glaforge.free.fr/weblog/index.php?itemid=99。

② http://grails.org

变为活跃的。

作为应用开发者，通过仔细应用动态语言的功能，我们可以更具开发效率。而更高的开发效率意味着可以在更短的时间内轻松创建更高层的抽象，也可以利用一组人数较少但更能干的开发者来创建应用。此外，更高的开发效率还意味着可以快速创建应用的某些部分，然后得到开发人员、测试人员、领域专家和客户代表等同仁的反馈，而这一切又会使我们更为敏捷。关于开发Web应用，Tim O'Reilly观察到："不同于完成的画作，它们（即Web应用）只是轮廓，作为对新数据的响应而不断重绘。"在"Why Scripting Languages Matter"（为什么脚本语言至关重要）一文中[①]，他也表达了动态语言更适合Web开发的观点。

动态语言已经存在了很长时间，那为什么现在让人倍感兴奋了呢？原因至少有四点：

- 机器速度
- 可用性
- 对单元测试的意识
- 杀手级应用

先从机器速度开始看。将其他语言在编译时做的事情拿到运行时做，这会引发人们对动态语言速度的担忧。在运行时解释代码，而不是简单地执行编译好的代码，也加剧了这种担忧。好在这些年来机器的速度一直在提升，今天手持设备的计算能力和内存都超过了几十年前的大型机。有些任务，使用20世纪80年代的处理器可能是难以想象的，但现在却可以轻而易举地实现。得益于处理器速度及本领域中其他方面的提升，包括更好的即时编译技术和JVM对动态语言的支持等，我们对动态语言性能的担忧已经大大缓解。

再来谈一下可用性。互联网和活跃的基于社区的"开放"开发方式，使较新的动态语言易于获得和使用。开发者可以轻松地下载到这些语言和工具，并加以研究及利用，他们甚至可以参与到社区论坛中来影响语言的演进。Groovy用户邮件列表非常活跃，经常有热心用户参与讨论，表达他们对当前和未来特性的意见、想法和批评。[②]这使我们能够比以往更好地实验、学习并调整语言。

下面再来看一下单元测试意识。大部分动态语言是动态类型的。[③]类型往往基于对上下文的推断。没有编译器在编译时标记类型强制转换违例。由于很多代码可能是在运行时合成的，而且程序可以在运行时扩展，所以不能单独依赖编写代码时的验证。从测试的角度看，相对于使用静态类型语言编写代码，使用动态语言需要更严格的自律。在过去的几年里，我们看到程序员对测试，特别是单元测试的意识在逐渐增强（尽管采用广度还远远不够）。大部分将这些动态语言应

① 文章链接：http://www.oreillynet.com/pub/wlg/3190。Tim O'Reilly探讨了应用的本质及脚本语言扮演的角色。

② 可以参阅http://groovy.codehaus.org/Mailing+Lists和http://groovy.markmail.org。

③ 这里需要明确，"动态语言"中的"动态"指的是前面提到的"将其他语言在编译时做的事情拿到运行时做"，而非"动态类型"中的"动态"。参见http://en.wikipedia.org/wiki/Dynamic_programming_language。——译者注

用于商业应用的程序员，已经采用了测试和单元测试。

最后，很多开发者实际上已经使用了几十年动态语言了。然而，要唤起业内大多数人对动态语言的兴趣，必须有可以与开发者和管理人员分享的杀手级应用，也就是有令人信服的应用案例。这一引爆点——往小了说是Ruby的，往大了说是动态语言的——以Rails[1]的形式出现了。Rails显示出，使用Ruby的动态功能，苦苦挣扎的Web开发者可以如何快速地开发应用。同样，我们有Grails[2]这种用Groovy和Java编写的Web框架，它提供了同样的开发效率和便利性。

这些框架在开发社区引起了足够的轰动，使得在整个业界应用动态语言有了极大的可能性。

动态语言，连同元编程功能，使简单的事情更简单，复杂的事情也可以掌控。当然我们仍然需要处理应用的内部复杂性，但动态语言让我们有可能把力气用在刀刃上。在使用了多年C++之后，当我接触到Java时，像反射、良好的类库以及不断演化的框架支持等特性带给我非常高的开发效率 。JVM在一定程度上为我提供了使用元编程的能力。然而，除了Java，我还不得不想办法利用一些可能较为重量级的工具，比如AspectJ。和其他一些开发效率很高的程序员一样，我发现自己有两条路可走：使用极为复杂而且不那么灵活的Java，结合多种重量级工具；或者转而使用面向对象的、内建元编程功能的动态语言，比如Ruby。（举个例子，在Ruby和Groovy中实现面向方面编程只需要几行代码。）在几年之前，保持高开发效率的同时利用动态功能和元编程，就意味着要离开Java平台。（毕竟，这些特性的使用是为了提高效率，不能让它们拖慢我们的脚步，对不对？）世易时移，情况变了。现在有了诸如Groovy、JRuby和Clojure这样的动态且运行在JVM上的语言。使用这些语言，我们可以充分利用Java平台的丰富特性和动态语言功能。

## 为何选择Groovy

作为Java程序员，我们不必完全切换到一门不同的语言。Groovy感觉就像我们已经熟知的Java外加一些扩展。

很多脚本语言都能在JVM上运行，如Groovy、JRuby、BeanShell、Scheme、Jaskell、Jython和JavaScript等，举不胜举。我们应该基于一系列标准来选择语言：需求、偏好和背景，开发的项目，以及公司的技术背景等。本节探讨一下Groovy何时是正确的选择。

因为下面一些原因，Groovy很有吸引力：

- 易于掌握
- 遵循 Java 语义
- 满足了我们对动态语言的热爱
- 扩展了 JDK

① http://rubyonrails.org
② http://grails.org

我们来详细研究一下。首先，几乎可以把任何Java代码当作Groovy代码来运行（有一些已知的存在问题的地方，参见2.11节），这意味着学习起来非常易于掌握。现在就可以开始在Groovy中编写代码了，如果卡壳，只需要换个思路，直接编写我们熟悉的Java代码。可以以后再重构那些代码，使其更符合Groovy风格。

例如，Groovy理解传统的`for`循环，因此可以这样写代码：

```
// Java风格
for(int i = 0; i < 10; i++) {
        //...
}
```

学习了Groovy后，可以将上面的代码修改为以下形式，或者修改为Groovy中其他风格的循环形式（现在不用担心语法，毕竟我们才刚刚起步，你很快就会成为这方面的专家）。

```
10.times {
         //...
}
```

其次，在使用Groovy编程时，Java有的Groovy几乎都有。Groovy类同样也扩展了古老的`java.lang.Object`类，Groovy类就是Java类。面向对象范型和Java语义也都保留了下来，所以在使用Groovy编写表达式和语句时，对于我们Java程序员而言，其实已经知道其意义。

这里有一个用以演示Groovy类就是Java类的小例子：

**Introduction/UseGroovyClass.groovy**

```
println XmlParser.class
println XmlParser.class.superclass
```

运行`groovy UseGroovyClass`，会得到如下输出：

```
class groovy.util.XmlParser
class java.lang.Object
```

爱上Groovy的第三个原因——Groovy是动态的，类型也是可选的。也许你已经在诸如Smalltalk、Python、JavaScript和Ruby等动态类型语言中体会过这些特性的好处，现在在Groovy中你也可以体会到。例如，要向`String`类添加`isPalindrome()`方法，以判断一个单词是否为回文结构，即正向拼写和逆向拼写是否一致，那非常容易，只需要几行代码（再次说明，现在不必尝试理解其工作原理的所有细节，本书其余部分会予以解决）：

**Introduction/Palindrome.groovy**

```
String.metaClass.isPalindrome = {->
    delegate == delegate.reverse()
}

word = 'tattarrattat'
```

```
println "$word is a palindrome? ${word.isPalindrome()}"
word = 'Groovy'
println "$word is a palindrome? ${word.isPalindrome()}"
```

通过输出来看看这段代码的效果：

```
tattarrattat is a palindrome? true
Groovy is a palindrome? false
```

这说明不需要侵入一个类的源代码，即可使用便捷的方法轻松扩展该类，即便是神圣不可侵犯的java.lang.String类。

最后，Java程序员在工作时严重依赖JDK和API，而这些在Groovy中仍然可以使用。此外，通过Groovy JDK（GDK），Groovy使用便捷方法和闭包支持扩展了JDK。下面是一个简单的例子，说明了GDK对java.util.ArrayList的扩展：

```
lst = ['Groovy', 'is', 'hip']
println lst.join(' ')
println lst.getClass()
```

从这段代码的输出可以确认用到了JDK，但是除此之外，我们还能使用Groovy添加的join()方法把ArrayList中的元素连接起来：

```
Groovy is hip
class java.util.ArrayList
```

Groovy扩充了我们所孰知的Java。如果项目团队熟悉Java，该组织的大部分项目中也使用了Java，而且有很多Java代码要集成和使用，那Groovy就是提高开发效率的极佳途径。

## 本书内容

本书是关于Groovy编程的，面向对JDK已经有所了解且有意学习Groovy语言及其动态能力的Java程序员，书中将结合很多实际的例子来探索Groovy语言的特性。希望本书能帮助程序员利用这门有趣且强大的语言快速提高开发效率。

本书主要内容组织为以下四个部分。

第一部分包括本书的前6章，这些章节关注Groovy相关的“为什么”，以及“是什么”，帮助我们适应Groovy常规编程的基础。本书是为有经验的Java程序员编写的，所以不会在诸如if语句是什么、如何编写if语句这种编程基础知识上浪费任何时间。相反，会直接深入介绍Groovy与Java的相似之处，以及Groovy的特性等主题。

第二部分包括第7章到第10章，我们将看到如何将Groovy应用于日常编码，包括使用XML、访问数据库以及使用多个Java/Groovy类和脚本，因此可以立即把Groovy应用于日常任务。我们还将探讨Groovy对JDK的扩展与补充，这样就可以兼顾Groovy和JDK的优势了。

第三部分包括第11章到第16章，将深入研究Groovy的元编程能力，Groovy真正的特色和优势就在于此，你还会学到如何利用其动态特性。本部分将从元对象协议（MetaObject Protocol，MOP）的基础入手，介绍如何在Groovy中完成类AOP操作，并探讨方法/属性的动态发现与分派。我们还将探索编译时元编程能力，同时看一下这种能力对于在编译阶段扩展与变换代码有何帮助。

第四部分包括最后3章，我们将应用Groovy元编程来创建和使用生成器及领域特定语言（DSL）。在Groovy中，因为其动态特性，单元测试是必要的。不过通过本部分的学习，你会发现单元测试也很容易，我们可以使用Groovy对Java和Groovy代码进行单元测试。

你正在阅读的是引言部分，下面分别介绍一下每章的内容。

在第1章中，我们将下载并安装Groovy，通过试用`groovysh`和`groovyConsole`来快速熟悉Groovy。我们还会看到Groovy的其他运行方式，包括从命令行运行和在IDE内运行。

在第2章中，我们将从熟悉的Java代码入手，然后将其重构为Groovy风格的代码。在快速浏览过可以改进日常Java编码的Groovy特性之后，我们再来谈一下Groovy对Java 5特性的支持。Groovy遵循Java语义，仅在少数情况下例外，我们将会探讨这些陷阱，以防遭遇时措手不及。

在第3章中，我们将看到Groovy的类型与Java的类型的异同，Groovy如何处理我们提供的类型信息，以及何时可以利用动态类型或可选类型。本章会介绍如何利用Groovy的动态类型、能力式设计（Design by Capability）以及多方法（multimethods）。对于可以应用动态类型但性能要求无法满足的任务，我们会看到如何通知Groovy对部分代码进行静态类型化处理。

在第4章中，我们将学到关于闭包这一激动人心的Groovy特性的一切，包括闭包是什么、如何工作、何时使用，以及如何使用等。Groovy闭包比简单的Lambda表达式更强大；它还使尾递归优化和记忆化（Memoization）更方便使用了。

在第5章中，我们将探讨Groovy的字符串、多行字符串的运用，以及Groovy对正则表达式的支持。

在第6章中，我们将探索Groovy对Java集合类——`List`和`Map`的支持，包括集合类上的各种便捷方法。看完这些之后，我们就再也不想以原来的方式使用集合类了。

Groovy包含并扩展了JDK。在第7章中，我们将探索GDK及其特性，同时看看Groovy对`Object`类及其他Java类的扩展。

Groovy对处理XML——包括解析和创建XML文档——也提供了良好的支持，在第8章中我们会看到相关介绍。

第9章介绍了Groovy对SQL的支持，这些支持会让数据库相关的编程变得轻松有趣。这一章将介绍迭代器、数据集，以及如何使用更简单的语法和闭包执行常规的数据库操作，甚至如何从Microsoft Excel文档中获取数据。

与Java的集成是Groovy的主要优势之一。在第10章中，我们将研究在Groovy和Java代码内与多个Groovy脚本、Groovy类和Java类紧密交互的不同方式。

一般而言，元编程是动态语言的最大优势之一，在Groovy中该优势尤为突出；借助该特性，我们可以在运行时检查类，以及动态分派方法调用。在第11章中，我们将从Groovy如何处理在Groovy对象和Java对象上的方法调用这些基础入手，探索Groovy对元编程的支持。

利用Groovy，可以使用`GroovyInterceptable`和`ExpandoMetaClass`类执行类AOP的方法拦截，第12章会予以介绍。

在第13章中，我们将深挖Groovy的元编程能力，学习如何在运行时注入方法。

在第14章中，我们将学习如何在运行时合成或生成动态方法。

第15章介绍了如何动态合成类、如何使用元编程委托方法调用，以及如何在前面3章介绍的元编程技术中做出选择。

Groovy的优点不止于运行时元编程，利用抽象语法树（Abstract Syntax Tree，AST）变换技术，Groovy在静态编译时也具有同样的优势，在第16章中我们会看到。

Groovy生成器是专门辅助为嵌套层次结构创建灵活接口的类。在第17章中，我们将探讨如何使用生成器，以及如何创建自己的生成器。

在Groovy中，单元测试并非奢侈品，也不是“有时间才做”的练习。Groovy的动态特性需要单元测试。幸运的是，Groovy让编写测试和创建模拟对象变得很容易，第18章中将详细介绍。我们还会尝试有助于我们使用Groovy对Java代码和Groovy代码进行单元测试的技巧。

利用第19章中介绍的技术，我们可以应用Groovy的元编程能力来构建内部的DSL。我们将从DSL的基础（包括其特点）入手，然后快速转入在Groovy中构建DSL的学习。

最后，在附录中，列出了本书所引用的Web文章和书籍。

## Groovy的变化

现今Groovy又有了长足的进步，本书将介绍的是最新的Groovy 2.1。以下是本书中可能对你有所帮助的更新。

- 你将学到 Groovy 2.*x* 的特性。
- 你将学到`@Delegate`、`@Immutable` 等 Groovy 代码的生成变换。
- 你将学到新的 Groovy 2.*x* 静态类型检查和静态编译工具的优点。
- 你将学到利用 Groovy 2.*x* 中对扩展模块的新支持来创建自己的扩展方法的一些技巧。
- Groovy 中的闭包非常优秀，你将学到它们对尾递归优化和记忆化的新支持。
- 你将学到如何有效地集成 Java 和 Groovy，如何从 Java 中传递 Groovy 闭包，甚至从 Java

中调用动态的 Groovy 方法。
- 你会看到为学习元编程 API 的增强而引入的例子。
- 你将学到如何使用 Mixin，以及如何使用它们实现一些优雅的模式。
- 除了运行时元编程，你还可以掌握编译时元编程和抽象语法树（AST）变换。
- 你将看到构建与读取 JSON 数据的具体细节。
- 再有，你将学到有助于流畅地创建 DSL 的 Groovy 语法。

## 目标读者

本书是为在Java平台上工作的开发者编写的。它最适合那些对Java语言有较深了解的开发人员和测试人员。懂得使用其他语言编程的开发者也可以使用本书，但是他们需要利用有助于深入理解Java和JDK的书籍来补补课，*Effective Java*[Blo08]和*Thinking in Java*[Eck06]就是非常不错的学习资源。

对Groovy有所了解的程序员可以使用本书来学习一些在其他地方无从得见的诀窍和技巧。另外，已经非常熟悉Groovy的程序员可能会发现本书能够用于培训和辅导组织中的开发者同仁。

## 在线资源

本书引用的网络资源都汇集在了附录A中。下面两个资源可以帮助你入门。

- Groovy 网站，可以下载本书使用的 Groovy 版本：http://groovy.codehaus.org。
- 本书在 Pragmatic Bookshelf 网站的官方主页：http://www.pragprog.com/titles/vslg2。你可以从这里下载所有的示例源代码文件，也可以通过在本书的论坛中提交勘误或发表意见来反馈问题。

如果你正在阅读的是本书的电子版，可以点击代码清单上的链接来查看或下载具体的例子。

## 致谢

过去的四年中，看着Groovy生态系统成长起来真是非常快乐。感谢Groovy团队开发了这门帮助程序员提高开发效率的同时还能享受快乐的语言及其相关工具。

我要感谢本书第1版的所有读者。特别感谢Norbert Beckers、Giacomo Cosenza、Jeremy Flowers、Ioan Le Gué、Fred Janon、Christopher M. Judd、Will Krespan、Jorge Lee、Rick Manocchi、Andy O'Brien、Tim Orr、Enio Pereira、David Potts、Srivaths Sankaran、Justin Spradlin、Fabian Topfstedt、Bryan Young和Steve Zhang，感谢他们花时间在本书勘误页面上报告错误。

衷心感谢和感激本书的技术审校人员。他们非常友好地奉献出自己的时间和精力，通读了书

中的概念，试验了书中的例子，并向我提供了非常有价值的反馈、修正和鼓励。感谢你们，Tim Berglund、Mike Brady、Hamlet D'arcy、Scott Davis、Jeff Holland、Michael Kimsal、Scott Leberknight、Joe McTee、Al Scherer和Eitan Suez。

还要感谢Guillaume Laforge的鼓励，感谢他拨冗撰写序言。Cédric Champeau和Chris Reigrut慷慨地快速通读了本书的beta版本，并且提供了有价值的反馈。非常感谢你们，谢谢。同时感谢Thilo Maier在本书的勘误页面上报告了一些错误。

特别感谢本书的编辑Brian Hogan，感谢他的复核、点评、建议和鼓励。在这一版的创作过程中，他提供了必要的指导。

感谢整个Pragmatic Programmers团队，感谢他们开发Groovy的这个版本，还要感谢他们在出版过程中提供的支持。

# 目　录

## 第三部分　MOP 与元编程



## 第四部分　使用元编程

# Part 1 第一部分

# Groovy 起步

本部分内容

# 第 1 章 起 步

在开始编写Groovy代码之前，首先需要安装Groovy。本章将介绍如何快速安装Groovy，并确保一切工作正常。现在处理好这些基础任务，有助于我们向后面更有意思的内容快速迈进。

## 1.1 安装 Groovy

获得一份稳定、可用的Groovy副本非常简单：只需访问Groovy主页http://groovy.codehaus.org，点击下载链接。可以看到有二进制版本和源代码版本供我们下载：如果想在本地构建Groovy，或者想研究其源代码，请下载源代码版本；否则请下载二进制版本。Windows用户也可以下载Windows安装程序版本。建议访问下载页面的同时，把Groovy的文档也下载下来。

对于加入了Groovy用户邮件列表的前卫程序员，前面提到的版本是不够的。他们需要的是Groovy语言实现[①]的最新预发布版本，可以从http://groovy.codehaus.org/Git获取快照版本。

还需要JDK 1.5或更高版本，所以请确保本地系统上已经安装了Java[②]。

下面来安装Groovy。

### 1.1.1 在Windows系统环境安装Groovy

我们可以使用针对Windows的一键式安装程序，运行安装程序并按照说明操作即可。希望对安装过程施加更多控制的程序员，可以使用二进制发布包。

下一步，必须设置`GROOVY_HOME`环境变量和路径。进入“控制面板”，打开“系统”，编辑“系统变量”。创建一个名为`GROOVY_HOME`的环境变量，然后将其设置为Groovy目录的位置（例如，我将其设置为`C:\programs\groovy\groovy-2.1.0`）。此外，将`%GROOVY_HOME%\bin`添加到`Path`环境变量中，以此把Groovy的`bin`目录加入到查找路径中。记得路径中的目录以分号（`;`）分隔。

再下一步，确认环境变量`JAVA_HOME`指向的是Java开发包（Java Development Kit，JDK）的

①这里指支撑Groovy语言的编译器和运行时等。——译者注

② http://java.sun.com/javase/downloads/index.jsp

位置。如果不是，请设置。

这就完成了在Windows系统环境的安装。记得关闭所有打开的命令行窗口，因为对环境变量的修改需要重启命令行窗口才会生效。在新的命令行窗口中，输入`groovy -v`，确保报告的是正确的版本。

### 1.1.2 在类Unix系统环境安装Groovy

解压下载的二进制发布包。同时访问http://groovy.codehaus.org/Download，查看是否有针对不同Unix变种的特殊发布版本和说明。将`groovy-2.1.0`目录移到想放的位置。例如，在我的Mac系统上，我将其放在了`/opt/groovy`目录。

下一步，设置`GROOVY_HOME`环境变量与路径。根据所用Shell的不同，需要编辑不同的配置文件。你可能已经知道去哪里找那些配置文件；如果不知道，请参考相应文档。我在OS X上使用bash，所以我编辑的是`~/.bash_profile`文件。在这个文件中，我添加了一项：`export GROOVY_HOME="/opt/groovy/groovy-2.1.0"`，以此设置环境变量`GROOVY_HOME`。我还把`$GROOVY_HOME/bin`添加到了`PATH`环境变量中。

再下一步，确认环境变量`JAVA_HOME`指向的是JDK目录所在位置；如果不是，请设置。``ls -l `which java` ``这条命令可以帮助确定Java的安装位置。

完成了Groovy的安装，就为使用这门语言做好了基本准备工作。关闭所有打开的终端窗口，因为对环境变量的修改需要重启终端才会生效。也可以使用`source`命令执行一下配置文件，但打开新窗口简单明了。在一个新的命令行窗口中，输入`groovy -v`命令，确保报告的是正确的版本。这就够了！

## 1.2 管理多个版本的 Groovy

面对不同的项目，经常需要使用同一语言的多个版本。如果不够小心，为项目管理正确的语言版本这个任务可能很快就会变成消耗时间的无底洞。GVM（Groovy enVironment Manager）不仅可以管理Groovy语言的版本，还可以管理与Groovy相关的库和工具（如Grails、Griffon和Gradle等）的版本。

这款工具很容易安装，而且支持各种*nix系统，在Windows系统环境也可以通过Cygwin支持[①]。一旦安装了GVM，只需简单地运行`gvm list groovy`命令，就可以看到可用的和已安装的Groovy语言版本。如果想使用Groovy的某个特定版本，也可以指定。例如，要运行本书中的示例，可以输入`gvm install groovy 2.1.1`命令。GVM之后会下载并安装该版本，以供使用。如果已经安装了Groovy的多个版本，要切换到2.1.1版本，则可以使用`gvm use groovy 2.1.1`命令。

① http://gvmtool.net

## 1.3　使用 groovysh

我们已经安装了Groovy，并且核对了版本，现在就来试用吧 。命令行工具groovysh是把玩Groovy最为快速的方式之一。打开一个终端窗口，输入groovysh，如下所示，我们会看到一个shell。输入一些Groovy代码，看看它是如何工作的。

```
> groovysh
Groovy Shell (2.1.1, JVM: 1.7.0_04-ea)
Type 'help' or '\h' for help.
--------------------------------------------------
groovy:000> Math.sqrt(16)
===> 4.0
groovy:000> println 'Test drive Groovy'
Test drive Groovy
===> null
groovy:000> String.metaClass.isPalindrome = {
groovy:001>   delegate == delegate.reverse()
groovy:002> }
===> groovysh_evaluate$_run_closure1@64b99636
groovy:000> 'mom'.isPalindrome()
===> true
groovy:000> 'mom'.l

lastIndexOf(   leftShift(     length()
groovy:000> 'mom'.l
```

groovysh是以交互方式尝试一些小型Groovy代码例子的好工具。它也可以用于在编码过程中实验一些代码。然而需要注意的是，groovysh有些特殊之处。如果在使用该命令时遇到问题，可以使用save命令把代码保存到一个文件中，然后尝试使用groovy命令从命令行运行，以避免任何与工具有关的问题。一按下回车键，groovysh命令就会编译并执行输入完的语句，打印代码执行过程中的所有输出，并打印这条语句的执行结果。

例如，输入Math.sqrt(16)，它会打印结果——4.0。然而，如果输入println 'Test drive Groovy'，打印的则是引号中的字符串，后面加上null，说明println()什么都没有返回。

也可以输入占据多行的代码，如果groovysh提示存在问题，只需要在每行后面加一个分号，就像定义动态方法isPalindrome()的那行代码一样。当我们输入一个类、一个方法，甚至一个if语句时，groovysh会等我们完成输入再执行那段代码。groovy:提示符后面的数字告诉我们，已经累积的要执行代码的行数。

如果不太确定要输入的命令，可以输入所知道的尽可能多的字符，然后按Tab键。shell会打印以我们输入的部分名字打头的可用方法，就像前面的groovysh交互式shell使用片段中，在输入'mom'.l后，按下Tab键，可以看到提示的内容。如果只输入一个英文句点（.）然后按下Tab键，shell会询问是否要显示所有可用方法。

输入help可以得到所支持命令的列表。可以使用向上箭头查看已经输入的命令，这对于重复前面的语句或命令非常有用。它甚至会记住我们在前面的调用中输入的命令。

使用完毕，输入exit退出该工具。

## 1.4 使用 groovyConsole

Groovy也有些偏爱图形用户界面（Graphical User Interface，GUI）的用户——只需要在Windows资源管理器中双击groovyConsole.bat（在%GROOVY_HOME%\bin目录下查找该文件）。类Unix系统的用户可以使用他们喜欢的文件或目录浏览工具双击groovyConsole可执行脚本。还可以在命令行输入groovyConsole来启动控制台GUI工具。控制台GUI会弹出来，如图1-1所示。

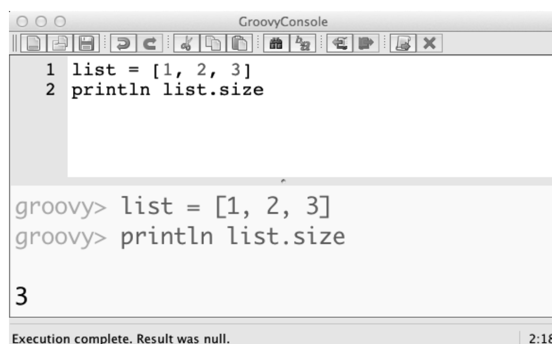



图1-1 使用groovyConsole

让我们在控制台的顶层窗口中输入一些Groovy代码。要执行代码，Windows系统用户按Ctrl+R或Ctrl+Enter组合键，Mac系统用户则按Command+R或Command+Enter组合键。

也可以通过点击相应的工具条按钮来执行脚本。随着时间的推移，groovyConsole变得越来越好了，可以执行保存脚本、打开现有脚本等操作，所以花点时间探索一下这个工具吧。

## 1.5 在命令行中运行 Groovy

当然，对某些程序员而言，再没有比进入命令行并运行程序更令他们开心的了。我们也可以这样做，只需输入groovy命令，后面加上Groovy程序的文件名，如下所示：

```
> cat Hello.groovy
println "Hello Groovy!"
> groovy Hello
Hello Groovy!
>
```

要在命令行中直接尝试一些语句，请使用-e选项。在命令行中输入groovy -e "println

`'hello'"`，并敲击回车，Groovy将输出“hello”。

然而，实际上`groovy`命令对于执行较大的Groovy脚本和类非常有用。它希望我们输入的或者是不包含在任何类中的一些可执行代码，或者是一个带有`static main(String[] args)`方法（即传统的Java `main()`方法）的类。

如果我们的类扩展了`GroovyTestCase`类（参见18.2节），或者实现了`Runnable`接口，可以跳过`main()`方法。在这些情况下，如果`main()`方法仍然出现了，则被优先执行。

## 1.6　使用 IDE

随着我们开始大量编写更为复杂的Groovy代码，我们将很快结束使用上述工具，并希望有一款功能齐全的集成开发环境（Integrated Development Environment，IDE）。幸运的是，我们有多种选择。参见http://groovy.codehaus.org/IDE+Support，这里提供了一些选择。应用这些工具，可以执行编辑Groovy代码、从IDE内运行代码、调试代码等操作（具体情况要取决于所选择的工具）。

### 1.6.1　IntelliJ IDEA

IntelliJ IDEA在免费的社区版中对Groovy提供了良好的支持[1]。通过IntelliJ IDEA，可以编辑Groovy代码，使用代码补全，获得对Groovy生成器的支持，利用语法和错误高亮，使用代码格式化与检查，联合编译Java和Groovy代码，重构与调试Java和Groovy代码，以及在同一项目中使用Java和Groovy代码。

### 1.6.2　Eclipse Groovy插件

Eclipse用户可以使用Groovy Eclipse插件[2]。通过该插件，可以编辑Groovy类和脚本，利用语法高亮，以及编译、运行与测试代码。使用Eclipse调试器，可以单步进入Groovy代码或调试单元测试。此外，还可以在Eclipse内调用Groovy Shell或Groovy控制台，以便快速实验Java和Groovy代码。

### 1.6.3　TextMate Groovy Bundle

Mac的程序员普遍是在TextMate中使用Groovy Bundle。关于TextMate，可以参考*TextMate: Power Editing for the Mac*[Gra07][3,4]一书。（Windows用户可以看一下E Text Editor[5]。另外，如果

---

① http://www.jetbrains.com/idea
② http://groovy.codehaus.org/Eclipse+Plugin
③ http://docs.codehaus.org/display/GROOVY/TextMate
④ http://macromates.com
⑤ https://github.com/etexteditor/e

编辑的是较小的代码片段，可以使用Notepad2[①]。）TextMate提供了一些可以节省时间的脚本片段，支持将一些代码展开为标准的Groovy代码，比如闭包。如图1-2所示，在TextMate内，我们可以利用语法高亮，还可以快速运行Groovy代码和测试。可以阅读我的博客文章http://blog.agiledeveloper. com/2007/10/tweaking-textmate-groovy-bundle.html来了解如何微调Groovy Bundle以快速显示结果而不弹出窗口。




图1-2　在TextMate内执行Groovy代码

很多TextMate用户正在转向新近出的编辑器Sublime Text。要在Sublime Text内运行Groovy代码，需要一个构建脚本。如果Tools > Build System菜单下没有，只要选择New Build System...菜单项创建一个名为`groovy.sublime-build`的文件，并在该文件中写入一行命令：

```
{ "cmd": ["/opt/groovy/bin/groovy", "$file"] }
```

该命令要求Sublime Text以Groovy源文件名为参数，运行指定路径上的`groovy`命令。结果将显示在输出窗口内。要运行代码，可以按F7或Command+B。有关Sublime Text中构建配置的更多细节，请参考http://sublimetext.info/docs/en/reference/build_systems.html。

有命令行和IDE工具可供选择真好。然而还需要确定哪款工具是正确的选择。我觉得最简单的方法是，直接在编辑器或IDE内运行Groovy代码，让`groovy`工具在背后处理代码的编译与执行。这对我的“快速编辑、编码和运行测试”这种开发循环很有帮助。有时我发现我会跳到`groovysh`实验一些代码片段。但是你不必这么做。自己用着最舒服的工具就是正确的。请从适合自己的简单工具和步骤入手。感觉适应了以后，当需要时可以尝试一些较为复杂的工具。

在本章中，我们安装了Groovy，并且进行了快速测试，还一并看了一些命令行工具和IDE支持。这意味着我们已经为下一章的探索做好了准备。

① http://www.flos-freeware.ch/notepad2.html

第2章

# 面向Java开发者的Groovy

因为Groovy支持Java语法，并且保留了Java语义，所以我们尽可以随心所欲地混用两种语言风格。本章将以我们熟悉的背景为起点，然后向更符合Groovy的编码风格过渡。即从过去常常使用Java完成的任务入手，随着将为完成这些任务而编写的Java代码逐步转变为Groovy代码，我们会看到Groovy版本更为简洁，而且更具表现力。在本章最后，我们会研究一些陷阱——如果预料不到，这些陷阱会让我们措手不及。

## 2.1　从 Java 到 Groovy

我们从一段带有一个简单循环的Java代码入手。首先通过Groovy运行这段代码，然后将其从Java风格重构为Groovy风格。在代码演进过程中，每个版本所做的事情相同，但是代码越来越简洁，表现力也越来越好。那感觉就像打了兴奋剂。我们开始吧。

### 2.1.1　Hello，Groovy

让我们从一个以Java代码编写的例子开始，它同时也是Groovy代码，保存在一个名为Greetings.groovy的文件中。

```
// Java代码
public class Greetings {
  public static void main(String[] args) {
    for(int i = 0; i < 3; i++) {
      System.out.print("ho ");
    }

    System.out.println("Merry Groovy!");
  }
}
```

使用groovy Greetings.groovy命令执行这段代码，看一下输出：

```
ho ho ho Merry Groovy!
```

这么简单的任务，代码可真够多的。不过Groovy依然乖乖地接受并执行了它。

Groovy的信噪比比Java要高，故而可以用较少的代码获得更多结果。实际上，去掉上面程序中的大部分代码，仍然可以得到相同的结果。我们就从去掉表示一行结束的分号开始。去掉分号能减少噪音，代码也会更流畅。

2

现在去掉类和方法定义，Groovy仍是欣然接受（说不定更喜欢了）。

**默认导入**

在编写Groovy代码时，不必导入所有的常用类或包。例如，使用Calendar，就可以毫无困难地引用java.util.Calendar。Groovy自动导入下列包：java.lang、java.util、java.io和java.net。它也会导入java.math.BigDecimal和java.math.BigInteger两个类。此外，它还导入了groovy.lang和groovy.util这些Groovy包。

**GroovyForJavaEyes/LightGreetings.groovy**

```
for(int i = 0; i < 3; i++) {
  System.out.print("ho ")
}

System.out.println("Merry Groovy!")
```

甚至可以更进一步。Groovy能够理解println()，因为该方法已经被添加到java.lang.Object中。它还有一种使用Range对象的、更为轻量级的for循环形式，而且Groovy对括号很宽容。因此，可以把上面代码简化成下面这样：

**GroovyForJavaEyes/LighterGreetings.groovy**

```
for(i in 0..2) { print 'ho ' }

println 'Merry Groovy!'
```

这段代码的输出与本节开始所编写的Java代码相同，但是代码轻便多了。在Groovy中，简单的事情就简单做。

### 2.1.2 实现循环的方式

Groovy并没有限制使用传统的for循环。我们已经在for循环中用过了range 0..2。其实，Groovy为我们提供了很多优雅的迭代方式。

比如upto()，它是Groovy向java.lang.Integer类中添加的一个便于使用的实例方法，可用于迭代。

**GroovyForJavaEyes/WaysToLoop.groovy**

```
0.upto(2) { print "$it "}
```

我们在0上调用了upto()，这里的0就是Integer的一个实例。输出应该显示所选范围内的每个值。

```
0 1 2
```

那代码块中的$it是什么呢？在这个上下文中，它代表进行循环时的索引值。upto()方法接受一个闭包作为参数。如果闭包只需要一个参数，在Groovy中则可以使用默认的名字it来表示该参数。（先记住这一点，第4章将更详细地讨论闭包。）变量it前面的$让print()方法打印该变量的值，而非打印it这两个字符。利用该特性，我们可以在字符串中嵌入表达式，第5章有此类用法。

使用upto()方法时，可以设置范围的上限和下限。如果范围从0开始，也可以使用times()，如下面的例子所示：

**GroovyForJavaEyes/WaysToLoop.groovy**

```
3.times { print "$it "}
```

这个版本将与上个版本产生相同的输出：

```
0 1 2
```

要在循环时跳过一些值，可以使用step()方法。

**GroovyForJavaEyes/WaysToLoop.groovy**

```
0.step(10, 2) { print "$it "}
```

输出将显示该范围内选定的值：

```
0 2 4 6 8
```

还可以使用类似方法迭代或遍历对象的集合，第6章将介绍相关用法。

接下来，使用前面学到的方法重写Greetings这个例子。与我们开始所写的Java代码相比，看看Groovy代码是多么简短吧：

**GroovyForJavaEyes/WaysToLoop.groovy**

```
3.times { print 'ho ' }
println 'Merry Groovy!'
```

为确认这段代码的效果，运行并查看输出：

```
ho ho ho Merry Groovy!
```

## 2.1.3　GDK一瞥

Java平台的核心优势之一就是其Java开发包（JDK）。Groovy并没有强迫我们学习一组新的类和库。通过向JDK的各种类中添加便捷方法，Groovy扩展了强大的JDK。这些扩展可以在称作GDK（或Groovy JDK，http://groovy.codehaus.org/groovy-jdk）的库中获得。通过使用Groovy提供的便捷方法，我们甚至可以更深入地利用JDK。下面通过一个用于与外部进程通信的GDK便捷方法来激发我们的兴趣。



我生命中有一部分时间花在了版本控制系统的维护上。每当有文件签入时，后端的钩子会使用一些规则，执行进程，然后发出通知。简而言之，我必须创建进程，并与这些进程交互。来看一下Groovy在这里是如何帮助我们的。

Java中可以使用`java.lang.Process`与系统级进程交互。假设我们想在代码中调用Subversion的`help`，下面是实现该功能的Java代码：

```
//Java代码
import java.io.*;
public class ExecuteProcess {
  public static void main(String[] args) {
    try {
      Process proc = Runtime.getRuntime().exec("svn help");
      BufferedReader result = new BufferedReader(
        new InputStreamReader(proc.getInputStream()));
      String line;
      while((line = result.readLine()) != null) {
        System.out.println(line);
      }
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

`java.lang.Process`类非常有用，但是在前面的代码中使用它时，真是大费周章；实际上，异常处理代码以及为实现输出所做的努力，所有这些让我们头晕目眩。通过在`java.lang.String`类上添加一个`execute()`方法，GDK使这一切变得非常简单了。

**GroovyForJavaEyes/Execute.groovy**

```
println "svn help".execute().text
```

比较这两个代码段，让我想起了电影《夺宝奇兵》中持剑打斗的场景。Java代码就像带剑的反派，搞了一个大大的噱头。而另一方面，Groovy就像印第安纳·琼斯博士，不费吹灰之力就把任务完成了。不要误解我的意思——我当然不是说Java是反派。在Groovy代码中，我们仍然使用了`Process`和JDK。那些让利用JDK和Java平台的力量更困难且更耗时的不必要的复杂性，才是我们的敌人。

在我维护的一个Subversion钩子中，一次重构把50多行Java代码减少到了只有3行Groovy代码。这两个版本我更喜欢哪一个呢？当然是简洁明了的那个了（除非我们是按代码行数收费的咨询师……）。

当在String实例上调用execute()方法时，Groovy创建了一个扩展了java.lang.Process的类的实例，就像Java代码中Runtime类的exec()方法所做的那样。可以使用如下代码验证这一点：

**GroovyForJavaEyes/Execute.groovy**

```
println "svn help".execute().getClass().name
```

当在类Unix机器上运行时，输出如下：

```
java.lang.UNIXProcess
```

在Windows机器上，输出则是：

```
java.lang.ProcessImpl
```

当调用text时，我们是在调用Groovy在Process类上添加的getText()方法，其功能是将该进程的整个标准输出读到一个String对象中。如果只是想等待进程结束，waitFor()或Groovy添加的waitForOrKill()方法（该方法接受一个以毫秒表示的超时值）会有所帮助。这就来试一下上面的代码吧。

除了使用Subversion，还可以尝试其他命令：只需要将svn help替换成其他程序，比如groovy-v。

**GroovyForJavaEyes/Execute.groovy**

```
println "groovy -v".execute().text
```

我们在Groovy脚本内调用的独立Groovy进程将报告其版本号：

**GroovyForJavaEyes/Execute.output**

```
Groovy Version: 2.1.1 JVM: 1.7.0_04-ea Vendor: Oracle Corporation OS: Mac OS X
```

这个例子在类Unix系统和Windows系统上均可工作。类似地，在一个类Unix系统上，要得到当前目录下内容的列表，可以调用ls：

**GroovyForJavaEyes/Execute.groovy**

```
println "ls -l".execute().text
```

在Windows上，简单地把ls替换为dir是不起作用的。原因在于，尽管ls是一个可以在类Unix系统上执行的程序，但dir并不是一个程序，它只是一个shell命令。所以除了调用dir，还得多干点活。明确地说，我们需要调用cmd，并让它来执行dir命令：

**GroovyForJavaEyes/Windows/ExecuteDir.groovy**

```
println "cmd /C dir".execute().text
```

我们已经看到GDK扩展能使我们的编码工作更为轻松，但是仅仅触及了GDK的一点皮毛。第7章将研究GDK的更多精彩内容。



### 2.1.4　安全导航操作符

Groovy有很多激动人心且能帮助简化开发工作的小特性，继续阅读本书，你会发现这些特性遍布各个章节。安全导航（safe-navigation）操作符（?.）就是其中之一。我们经常需要检查引用是否为空值（null）。这种操作单调乏味，如下面例子所示，使用该操作符，可以避免这种操作：

**GroovyForJavaEyes/Ease.groovy**

```
def foo(str) {
  //if (str != null) { str.reverse() }
  str?.reverse()
}

println foo('evil')
println foo(null)
```

foo()方法（介绍编程的书籍中，往往至少会出现一个名为foo的方法）中的?.操作符只有在引用不为null时才会调用指定的方法或属性。运行这段代码，看一下输出：

```
live
null
```

使用?.在空引用上调用reverse()，其结果是产生了一个null，而没有抛出空指针异常（NullPointerException），这是Groovy减少噪音、节省开发者力气的另一手段。

### 2.1.5　异常处理

与Java相比，Groovy少了很多繁文缛节。这一点在异常处理上极其明显。Java强制我们处理所有受检查异常（Checked Exception）。试想一个简单的例子：我们想调用Thread的sleep()方法。（Groovy提供了一个备选的sleep()方法，参见7.1.3节。）Java坚持让我们捕获java.lang.InterruptedException。当不得已为之时，Java程序员会怎么做呢？想办法处理呗。结果就是大量空的catch块，是不是？看看这个：

**GroovyForJavaEyes/Sleep.java**

```
// Java代码
try {
```

```
    Thread.sleep(5000);
} catch(InterruptedException ex) {
    // 啊？这里该做什么？我都因为这个寝食难安了。
}
```

使用空的catch块比不处理异常更糟糕。如果放一个空的catch块，异常就被压制了下来。而如果我们在异常出现的第一个位置不予处理，该异常会被传递给调用者。这样调用者就可以做些处理工作，或是将其再传递给更上层的调用者。

对于那些我们不想处理，或者不适合在代码的当前层次处理的异常，Groovy并不强制我们处理。我们不处理的任何异常都会被自动传递给更高一层。下面用一个例子来说明一下，Groovy针对异常处理给出的答案：

**GroovyForJavaEyes/ExceptionHandling.groovy**

```
def openFile(fileName) {
  new FileInputStream(fileName)
}
```

openFile()方法没有处理声名狼藉的FileNotFoundException异常。如果产生了该异常，它并不会被压制下来。相反，它会被传递给调用代码，由调用代码来处理，如下面的例子所示：

**GroovyForJavaEyes/ExceptionHandling.groovy**

```
try {
  openFile("nonexistentfile")
} catch(FileNotFoundException ex) {
  // 关于该异常，在这里想做什么就做什么
  println "Oops: " + ex
}
```

如果有兴趣捕获可能抛出的所有异常（Exception），可以简单地在catch语句中省略异常的类型：

**GroovyForJavaEyes/ExceptionHandling.groovy**

```
try {
  openFile("nonexistentfile")
} catch(ex) {
  // 关于该异常，在这里想做什么就做什么
  println "Oops: " + ex
}
```

利用catch(ex)（变量ex前面没有任何类型）可以捕获摆在我们面前的任何异常。注意：它不能捕获Exception之外的Error或Throwable。要捕获所有这些，请使用catch(Throwable throwable)。

可见，Groovy让我们专注于完成工作，而不是将精力浪费在处理恼人的系统级细节上。

### 2.1.6 Groovy是轻量级的Java

Groovy还有其他一些使这门语言更为轻量级、更为易用的特性，试举几例如下。

- `return`语句几乎总是可选的（参见2.11.1节）。
- 尽管可以使用分号分隔语句，但它几乎总是可选的（参见2.11.6节）。
- 方法和类默认是公开（`public`）的。
- `?.`操作符只有对象引用不为空时才会分派调用。
- 可以使用具名参数初始化JavaBean（参见2.2节）。
- Groovy不强迫我们捕获自己不关心的异常，这些异常会被传递给代码的调用者。
- 静态方法内可以使用`this`来引用`Class`对象。在下面的例子中，`learn()`方法返回的是`Class`对象，所以可以使用链式调用：

```
class Wizard {
  def static learn(trick, action) {
    //...
    this
  }
}
Wizard.learn('alohomora', {/*...*/})
 .learn('expelliarmus', {/*...*/})
 .learn('lumos', {/*...*/})
```

在领教了Groovy的表现力和简洁这两大特性之后，下一节将介绍Groovy是如何在Java的一大最基本特性中减少混乱的。

## 2.2 JavaBean

JavaBean概念的引入令我们异常兴奋，那些能够按照特定约定暴露出其属性的Java对象就被视作JavaBean。这一概念唤起了很多希望，但是不久我们发现，要访问这些属性，开发者还是需要调用访问器（Getter）和更改器（Setter）方法。我们的兴奋轰然倒塌，而开发者还是要继续在其应用中创建成千上万傻瓜似的方法[1]。如果把JavaBean比作人，那么他们应该吃百忧解[2]了。公平地说，`JavaBean`的意图是好的，它使基于组件的开发、应用装配和集成变得切实可行，也为优秀的集成开发环境（IDE）和插件开发铺平了道路。

在Groovy中，JavaBean获得了应有的尊重。而且Groovy中的JavaBean确实是有属性的。我们就从Java代码入手，然后将其简化成Groovy代码，这样就能看到差别了。

---

① http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html
② 一种治疗精神抑郁的药物。——编者注

**GroovyForJavaEyes/Car.java**

```
//Java代码
public class Car {
  private int miles;
  private final int year;

  public Car(int theYear) { year = theYear; }
  public int getMiles() { return miles; }
  public void setMiles(int theMiles) { miles = theMiles; }

  public int getYear() { return year; }

  public static void main(String[] args) {
    Car car = new Car(2008);

    System.out.println("Year: " + car.getYear());
    System.out.println("Miles: " + car.getMiles());
    System.out.println("Setting miles");
    car.setMiles(25);
    System.out.println("Miles: " + car.getMiles());
  }
}
```

这是我们再熟悉不过的Java代码，不是吗？看一下Car实例属性的输出：

```
Year: 2008
Miles: 0
Setting miles
Miles: 25
```

前面的Java代码可以在Groovy中运行，但是如果用Groovy重写，则可以去掉很多乱七八糟的东西：

**GroovyForJavaEyes/GroovyCar.groovy**

```
class Car {
  def miles = 0
  final year

  Car(theYear) { year = theYear }
}

Car car = new Car(2008)

println "Year: $car.year"
println "Miles: $car.miles"
println  'Setting miles'
car.miles = 25
println "Miles: $car.miles"
```

这段代码和前面的Java代码做的事情是一样的（从输出可以看出），但是少了很多混乱和繁文缛节。

```
Year: 2008
Miles: 0
Setting miles
Miles: 25
```

def在这个上下文中声明了一个*属性*。我们可以像例子中这样使用def声明属性，还可以像int miles或int miles = 0这样给出类型（以及可选的值）。Groovy会在背后默默地为其创建一个访问器和一个更改器（就像在Java中，如果没有编写任何构造器，则Java编译器会创建一个）。当在代码中调用miles时，其实并非引用一个字段，而是调用该属性的访问器。要把属性设置为只读的，需要使用final来声明该属性，这和Java中一样。在这种情况下，Groovy会为该属性提供一个访问器，但不提供更改器。修改final字段的任何尝试都会导致异常。可以根据需要向声明中加入类型信息。可以把字段标记为private，但是Groovy并不遵守这一点[①]。因此，如果想把变量设置为私有的，必须实现一个拒绝任何修改的更改器。可以通过下面的代码验证这些概念：

**GroovyForJavaEyes/GroovyCar2.groovy**

```
class Car {
  final year
  private miles = 0

  Car(theYear) { year = theYear }

  def getMiles() {
    println "getMiles called"
    miles
  }

  private void setMiles(miles) {
    throw new IllegalAccessException("you're not allowed to change miles")
  }

  def drive(dist) { if (dist > 0) miles += dist }
}
```

这里使用final声明了year，使用private声明了miles。在drive()实例方法中，无法修改year，但是可以修改miles。miles的更改器不允许在类的外部对该属性的值进行任何修改。下面来使用这个版本的Car类。

① Groovy的实现不区分public、private和protected，参见http://groovy.codehaus.org/Things+you+can+do+but+better+ leave+undone。——译者注

**GroovyForJavaEyes/GroovyCar2.groovy**

```
def car = new Car(2012)

println "Year: $car.year"
println "Miles: $car.miles"
println  'Driving'
car.drive(10)
println "Miles: $car.miles"

try {
  print 'Can I set the year? '
  car.year = 1900
} catch(groovy.lang.ReadOnlyPropertyException ex) {
  println ex.message
}

try {
  print 'Can I set the miles? '
  car.miles = 12
} catch(IllegalAccessException ex) {
  println ex.message
}
```

从输出可以看到，我们能够读取两个属性的值，但是不能设置其中任何一个。

```
Year: 2012
getMiles called
Miles: 0
Driving
getMiles called
Miles: 10
Can I set the year? Cannot set readonly property: year for class: Car
Can I set the miles? you're not allowed to change miles
```

要想存取属性，再也不需要在调用中使用访问器和更改器了。下面代码说明了其优雅：

**GroovyForJavaEyes/UsingProperties.groovy**

```
Calendar.instance
// 代替Calendar.getInstance()
str = 'hello'

str.class.name
// 代替str.getClass().getName()
// 注意：不能用于Map、Builder等类型
// 为保险起见，请使用str.getClass().name
```

然而请谨慎使用`class`属性，像`Map`、生成器等一些类对该属性有特殊处理（参见6.5节）。因此，为避免任何意外，一般使用`getClass()`，而不是`class`。

## 2.3 灵活初始化与具名参数

Groovy中可以灵活地初始化一个JavaBean类。在构造对象时，可以简单地以逗号分隔的名值对来给出属性值。如果类有一个无参构造器，该操作会在构造器之后执行。[①] 也可以设计自己的方法，使其接受具名参数。要利用这一特性，需要把第一个形参定义为Map。下面通过代码来实际地看一下。

2

**GroovyForJavaEyes/NamedParameters.groovy**

```
class Robot {
  def type, height, width
  def access(location, weight, fragile) {
    println "Received fragile? $fragile, weight: $weight, loc: $location"
  }
}
robot = new Robot(type: 'arm', width: 10, height: 40)
println "$robot.type, $robot.height, $robot.width"

robot.access(x: 30, y: 20, z: 10, 50, true)
//可以修改参数顺序
robot.access(50, true, x: 30, y: 20, z: 10)
```

运行上面代码，看一下输出：

```
arm, 40, 10
Received fragile? true, weight: 50, loc: [x:30, y:20, z:10]
Received fragile? true, weight: 50, loc: [x:30, y:20, z:10]
```

Robot实例把type、height和width等实参当作了名值对。这里使用了Groovy编译器为我们创建的灵活的构造器。

access()方法有3个形参，但如果第一个是Map，则可以将这个映射中的键值对展开放在放在实参列表中。在第一次调用access()方法时，我们依次放了这个映射、weight以及fragile的值。不过如果我们愿意，用于这个映射的实参可以继续往后移，就像第2次调用access()方法那样。

如果发送的实参的个数多于方法的形参的个数，而且多出的实参是名值对，那么Groovy会假设方法的第一个形参是一个Map，然后将实参列表中的所有名值对组织到一起，作为第一个形参的值。之后，再将剩下的实参按照给出的顺序赋给其余形参，正如我们在输出中看到的那样。

尽管在Robot的例子中，这种灵活性非常强大，但是可能会给人带来困惑，所以请谨慎使用。如果想使用具名参数，那最好只接受一个Map形参，而不要混用不同的形参。在这个例子中，如

① 要使此类操作正确执行，类中必须有一个无参构造器。在示例代码中，因为没有定义构造器，编译器会提供一个无参的构造器。如果定义了带参数的构造器，则编译器不会再为我们创建无参构造器，所以一定要记得自己提供。读者可以自行修改代码测试。——译者注

果传递的是3个整型实参，该特性会导致一个问题。这种情况下，编译器将按顺序传递实参，而不会从这些实参创建一个映射，因而结果也就不是我们想要的了。

通过显式地将第一个形参指定为Map，可以避免这种混乱：

**GroovyForJavaEyes/NamedParameters.groovy**

```
def access(Map location, weight, fragile) {
  print "Received fragile? $fragile, weight: $weight, loc: $location"
}
```

现在，如果实参包含的不是两个对象外加一个任意的名值对，代码就会报错。

正如我们所见，由于Groovy给JavaBean换上了新装，JavaBean在Groovy中又生机勃发了。

## 2.4　可选形参

Groovy中可以把方法和构造器的形参设为可选的。实际上，我们想设置多少就可以设置多少，但这些形参必须位于形参列表的末尾。利用这一特性，可以在演进式设计中向已有方法添加新的形参。

要定义可选形参，只需要在形参列表中给它赋上一个值。下面是一个例子，log()方法带有一个可选的base形参。如果调用时不提供相应实参，则Groovy会假定其值为10：

**GroovyForJavaEyes/OptionalParameters.groovy**

```
def log(x, base=10) {
  Math.log(x) / Math.log(base)
}

println log(1024)
println log(1024, 10)
println log(1024, 2)
```

如输出所示，Groovy使用这个可选的值填充了缺失的实参：

```
3.0102999566398116
3.0102999566398116
10.0
```

Groovy还会把末尾的数组形参视作可选的。所以在下面的例子中，可以为最后一个形参提供零个或多个值：

**GroovyForJavaEyes/OptionalParameters.groovy**

```
def task(name, String[] details) {
  println "$name - $details"
```

```
}

task 'Call', '123-456-7890'
task 'Call', '123-456-7890', '231-546-0987'
task 'Check Mail'
```

从输出可以看出，Groovy会把末尾的实参收集起来，赋给数组形参：

```
Call - [123-456-7890]
Call - [123-456-7890, 231-546-0987]
Check Mail - []
```

多次调用某个方法时，如果需要提供相同的实参，这会让人厌烦。可选的形参减少了噪音，而且允许提供合理的默认值。

## 2.5　使用多赋值

向方法传递多个参数，这在很多编程语言中都司空见惯。但是从方法返回多个结果，尽管可能非常实用，却不那么常见。要想从方法返回多个结果，并将它们一次性赋给多个变量，我们可以返回一个数组，然后将多个变量以逗号分隔，放在圆括号中，置于赋值表达式左侧即可。

后面的例子中有一个负责将全名分割为名字（First Name）和姓氏（Last Name）的方法。不出所料，`split()`方法就返回一个数组。可以把`splitName()`的结果赋给一对变量：`firstName`和`lastName`。Groovy会把结果中的两个值分别赋给这两个变量。

**GroovyForJavaEyes/MultipleAssignments.groovy**

```
def splitName(fullName) { fullName.split(' ') }

def (firstName, lastName) = splitName('James Bond')
println "$lastName, $firstName $lastName"
```

要将结果设置到两个变量中，不必创建临时变量并编写多条赋值语句，如输出所示：

```
Bond, James Bond
```

还可以使用该特性来交换变量，无需创建中间变量来保存被交换的值，只需将欲交换的变量放在圆括号内，置于赋值表达式左侧，同时将它们以相反顺序放于方括号内，置于右侧即可。

**GroovyForJavaEyes/MultipleAssignments.groovy**

```
def name1 = "Thomson"
def name2 = "Thompson"

println "$name1 and $name2"
(name1, name2) = [name2, name1]
println "$name1 and $name2"
```

从输出中可以看到，name1和name2的值被交换了。

```
Thomson and Thompson
Thompson and Thomson
```

我们已经看到，当赋值表达式左侧的变量与右侧的值数目相同时，Groovy是如何处理多赋值的。而当变量与值的数目不匹配时，Groovy也可以优雅地处理。如果有多余的变量，Groovy会将它们设置为null，多余的值则会被丢弃。

如下面例子所示，还可以指定多赋值中定义的单个变量的类型。下面使用来自著名卡通系列《猫和老鼠》中的动物来说明这一点。

**GroovyForJavaEyes/MultipleAssignments.groovy**

```groovy
def (String cat, String mouse) = ['Tom', 'Jerry', 'Spike', 'Tyke']
println "$cat and $mouse"
```

左侧只有两个变量，所以狗狗Spike和Tyke将被丢弃。

```
Tom and Jerry
```

**GroovyForJavaEyes/MultipleAssignments.groovy**

```groovy
def (first, second, third) = ['Tom', 'Jerry']
println "$first, $second, and $third"
```

第三个变量的值将被设置为null。

```
Tom, Jerry, and null
```

如果多余的变量是不能设置为null的基本类型，Groovy将抛出一个异常。这是一种新行为，在Groovy 2.*x*中，只要可能，int会被看作基本类型，而非Integer。

可见，Groovy使发送和接收多个参数变得非常容易了。

## 2.6　实现接口

在Groovy中，可以把一个映射或一个代码块转化为接口，因此可以快速实现带有多个方法的接口。本节，你将看到Java实现接口的方式，然后学习如何利用Groovy实现接口。

下面是用于向Swing的JButton注册事件处理器的Java代码，我们再熟悉不过了。调用addActionListener()时，需要一个实现了ActionListener接口的实例。其中创建了一个实现了该接口的匿名内部类，同时提供了所需的actionPerformed()方法。该方法要求提供一个ActionEvent实例作为参数，即便我们在这个例子中并未用到。

```java
// Java代码
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
```

```
    JOptionPane.showMessageDialog(frame, "You clicked!");
  }
});
```

Groovy提供了一个与之不同的惯用法，非常迷人，不需要actionPerformed()方法声明，也不需要显式地用new来创建匿名内部类的实例。

```
button.addActionListener(
  { JOptionPane.showMessageDialog(frame, "You clicked!") } as ActionListener
)
```

调用了addActionListener方法，同时为该方法提供了一个代码块，借助as操作符，相当于实现了ActionListener接口。

就是它了！Groovy自会处理剩下的工作。它会拦截对接口中任何方法的调用（这个例子中就是actionPerformed()），然后将调用路由到我们提供的代码块。要运行这段代码，还需要创建窗体（Frame）及其组件。完整的代码清单参见本节末尾。

对于有多个方法的接口，如果打算为其所有方法提供一个相同的实现，和上面一样，不需要特殊的操作。

假设想实现一个功能，随着鼠标被点击或者在应用中移动而显示鼠标指针的位置。在Java中，我们必须实现MouseListener和MouseMotionListener接口中的总共7个方法。因为Groovy对所有这些方法的实现都是相同的，所以它给我们带来了方便。

```
displayMouseLocation = { positionLabel.setText("$it.x, $it.y") }
frame.addMouseListener(displayMouseLocation as MouseListener)
frame.addMouseMotionListener(displayMouseLocation as MouseMotionListener)
```

这段代码创建了变量displayMouseLocation，它指向的是一个代码块。使用as操作符将其转化了2次，分别转化为MouseListener和MouseMotionListener。Groovy又一次处理了剩下的事情，我们从而可以把更多精力转向其他事情了。这里用了3行代码，而不会像在Java中那样用掉……不好意思，我又数行数了。

前面的例子中又出现了it变量。it表示方法的参数。如果正在实现的一个接口中的方法需要多个参数，那么可以将其分别定义为独立的参数，也可以定义为一个数组类型的参数，具体情况将在第4章讨论。

Groovy没有强制实现接口中的所有方法：可以只定义自己关心的，而不考虑其他方法。如果剩下的方法从来不会被调用，那也就没必要去实现这些方法了。当在单元测试中通过实现接口来模拟某些行为时，这项技术非常有用。

好了，这挺不错的，但是在大多数实际情况下，接口中的每个方法需要不同的实现。不用担心，Groovy可以摆平。只需要创建一个映射，以每个方法的名字作为键，以方法对应的代码体作为键值，同时使用简单的Groovy风格，用冒号（:）分隔方法名和代码块即可。此外，不必实现

所有方法，只需实现真正关心的那些即可。如果未予实现的方法从未被调用过，那么也就没有必要浪费精力去实现这些伪存根。当然，如果没提供的方法被调用了，则会出现NullPointerException。下面把这些内容放到一个例子里看看：

```
handleFocus = [
  focusGained : { msgLabel.setText("Good to see you!") },
  focusLost : { msgLabel.setText("Come back soon!") }
]
button.addFocusListener(handleFocus as FocusListener)
```

每当例子中的按钮获得焦点时，与focusGained键关联的第一个代码块就会被调用。当按钮失去焦点时，与focusLost键关联的代码块则会被调用。在这种情况下，这里的键相当于FocusListener接口中的方法。

如果知道所实现接口的名字，使用as操作符即可，但如果应用要求的行为是动态的，而且只有在运行时才能知道接口的名字，又该如何呢？asType()方法可以帮忙。通过将欲实现接口的Class元对象作为一个参数发送给asType()，可以把代码块或映射转化为接口。我们来看一个例子。

```
events = ['WindowListener', 'ComponentListener']
//上面的列表可能是动态的，而且可能来自某些输入
handler = { msgLabel.setText("$it") }
for (event in events) {
  handlerImpl = handler.asType(Class.forName("java.awt.event.${event}"))
  frame."add${event}"(handlerImpl)
}
```

想实现的接口（也就是想处理的事件）在列表events中。该列表是动态的，假设它会在代码执行期间通过输入来填充。事件公共的处理器位于变量handler指向的代码块中。我们对事件进行循环，对于每个事件，都使用了asType()方法为该接口创建了一个实现。在代码块上调用asType()方法，同时把使用forName()方法获得的、该接口的Class元对象传给它。一旦手头有了监听器接口的实现，就可以通过调用相应的add方法（如addWindowListener()）来注册该实现。调用add方法本身就是动态的。11.2节将介绍这些方法的更多细节。

上面的代码使用了asType()方法。如果不同方法有不同实现，就会使用一个映射代替单个代码块。在那种情况下，可以用类似的方式在映射上调用asType()方法。最后，如约分享本节开发的Groovy Swing代码的完整清单。

**GroovyForJavaEyes/Swing.groovy**

```
import javax.swing.*
import java.awt.*
import java.awt.event.*

frame = new JFrame(size: [300, 300],
  layout: new FlowLayout(),
```

```
  defaultCloseOperation: javax.swing.WindowConstants.EXIT_ON_CLOSE)
button = new JButton("click")
positionLabel = new JLabel("")
msgLabel = new JLabel("")
frame.contentPane.add button
frame.contentPane.add positionLabel
frame.contentPane.add msgLabel

button.addActionListener(
  { JOptionPane.showMessageDialog(frame, "You clicked!") } as ActionListener
)

displayMouseLocation = { positionLabel.setText("$it.x, $it.y") }
frame.addMouseListener(displayMouseLocation as MouseListener)
frame.addMouseMotionListener(displayMouseLocation as MouseMotionListener)

handleFocus = [
  focusGained : { msgLabel.setText("Good to see you!") },
  focusLost : { msgLabel.setText("Come back soon!") }
]
button.addFocusListener(handleFocus as FocusListener)
events = ['WindowListener', 'ComponentListener']
// 上面的列表可能是动态的，而且可能来自某些输入
handler = { msgLabel.setText("$it") }
for (event in events) {
  handlerImpl = handler.asType(Class.forName("java.awt.event.${event}"))
  frame."add${event}"(handlerImpl)
}

frame.show()
```

现在已经介绍完了Groovy实现接口的方式。它使注册事件和传递接口的匿名实现变得非常简单。将代码块和映射转化为接口实现也相当省时。

## 2.7 布尔求值

Groovy中的布尔求值与Java不同。根据上下文，Groovy会自动把表达式计算为布尔值。

来看一个具体的例子，下面的Java代码是不工作的：

```
//Java代码
String obj = "hello";
int val = 4;
if (obj) {} //错误
if(val) {} //错误
```

Java要求if语句的条件部分必须是一个布尔表达式，比如前面例子中的`if(obj != null)`和`if(val > 0)`。

Groovy可没那么挑剔。它会尝试推断，所以我们需要知道它是怎么思考问题的。

如果在需要布尔值的地方放了一个对象引用，Groovy会检查该引用是否为null。它将null视作false，将非null的值视作true，如以下代码所示：

```
str = 'hello'
if (str) { println 'hello' }
```

如输出所示，Groovy会将该表达式计算作布尔值：

```
hello
```

必须承认，前面关于true的说法并不完全正确。如果对象引用不为null，表达式的结果还与对象的类型有关。例如，如果对象是一个集合（如java.util.ArrayList），那么Groovy会检查该集合是否为空。因此，在这种情况下，只有当obj不为null，而且该集合至少包含一个元素时，表达式if(obj)才会被计算为true；请看下面代码示例的输出：

```
lst0 = null
println lst0 ? 'lst0 true' : 'lst0 false'
lst1 = [1, 2, 3]
println lst1 ? 'lst1 true' : 'lst1 false'
lst2 = []
println lst2 ? 'lst2 true' : 'lst2 false'
```

对于集合类（Collection），Groovy是如何将其处理为布尔值的呢？可以通过这段代码的输出来看看我们的理解是否正确：

```
lst0 false
lst1 true
lst2 false
```

集合类不是唯一受到特殊对待的。那么有哪些类型将被特殊对待，Groovy又是如何计算它们的呢？请参考表2-1。

**表2-1　类型与布尔求值对它们的特殊处理**

| 类　型 | 为真的条件 |
|---|---|
| Boolean | 值为true |
| Collection | 集合不为空 |
| Character | 值不为0 |
| CharSequence | 长度大于0 |
| Enumeration | Has More Elements()为true |
| Iterator | hasNext()为true |
| Number | Double值不为0 |
| Map | 该映射不为空 |
| Matcher | 至少有一个匹配 |
| Object[] | 长度大于0 |
| 其他任何类型 | 引用不为null |

除了使用Groovy内建的布尔求值约定，在自己的类中，还可以通过实现asBoolean()方法来编写自己的布尔转换。

## 2.8　操作符重载



Groovy支持操作符重载，可以巧妙地应用这一点来创建DSL（领域特定语言，参见第19章）。Java是不支持操作符重载的，那Groovy又是如何做到的呢？其实很简单：每个操作符都会映射到一个标准的方法[①]。在Java中，可以使用那些方法；而在Groovy中，既可以使用操作符，也可以使用与之对应的方法。

下面是一个演示操作符重载的例子：

**GroovyForJavaEyes/OperatorOverloading.groovy**

```
for(ch = 'a'; ch < 'd'; ch++) {
  println ch
}
```

我们通过++操作符实现了从字符a到c的循环。该操作符映射的是String类的next()方法，输出如下：

```
a
b
c
```

Groovy中还可以使用简洁的for-each语法，不过两种实现都用到了String类的next()方法：

**GroovyForJavaEyes/OperatorOverloading.groovy**

```
for (ch in 'a'..'c') {
  println ch
}
```

String类重载了很多操作符，5.4节将予以介绍。类似地，为方便使用，集合类（如ArrayList和Map）也重载了一些操作符。

要向集合中添加元素，可以使用<<操作符，该操作符会被转换为Groovy在Collection上添加的leftShift()方法，如下所示：

**GroovyForJavaEyes/OperatorOverloading.groovy**

```
lst = ['hello']
lst << 'there'
println lst
```

① http://groovy.codehaus.org/Operator+Overloading

在完成追加元素后，可以在输出中看到完整的集合：

```
[hello, there]
```

通过添加映射方法，我们可以为自己的类提供操作符，比如为+操作符添加plus()方法。

下面来为一个类添加一个重载的操作符：

**GroovyForJavaEyes/OperatorOverloading.groovy**

```
class ComplexNumber {
  def real, imaginary
  def plus(other) {
    new ComplexNumber(real: real + other.real,
          imaginary: imaginary + other.imaginary)
  }
  String toString() { "$real ${imaginary > 0 ? '+' : ''} ${imaginary}i"}
}
c1 = new ComplexNumber(real: 1, imaginary: 2)
c2 = new ComplexNumber(real: 4, imaginary: 1)
println c1 + c2
```

ComplexNumber类重载了+操作符。对于计算涉及负数平方根的复杂方程式，复数非常有用。复数有实部和虚部，就像人们的收入有实际收入和所得税申报单上的收入之分一样。因为在ComplexNumber类上添加了plus()方法，所以可以使用+操作符把两个复数加到一起，得到又一个作为结果的复数：

```
5 + 3i
```

当应用于某个上下文时，操作符重载可以使代码更富于表现力。应该只重载那些能使事物变得显而易见的操作符。例如，如果对于有上下文背景或领域知识的某些人而言，有些操作符反而不是那么直观，那重载可能就不是很好的选择。

在重载时，必须保留预期的语义。例如，+操作符不可以修改操作中的任何一个操作数。如果操作符必须是可交换的、对称的或传递的，则必须确保重载的方法遵循这些特性。

## 2.9　对 Java 5 特性的支持

像枚举和注解等Java 5的语言特性在Groovy中也可以工作。这意味着我们可以非常自然地混用Java和Groovy。回忆一下，Java 5引入了如下语言特性：

- 自动装箱
- for-each循环
- enum
- 变长参数（varargs）

- 注解
- 静态导入
- 泛型

下面来讨论一下Groovy对这些特性的支持程度。



### 2.9.1 自动装箱

因为Groovy具有动态类型特性，所以它从一开始就支持自动装箱。实际上，必要时Groovy会自动将基本类型视作对象。例如，执行下面代码：

**GroovyForJavaEyes/NotInt.groovy**

```
int val = 5

println val.getClass().name
```

报告的类型如下：

```
java.lang.Integer
```

在这段代码中，尽管指定的是`int`，但创建的是`java.lang.Integer`类的实例，而不是基本类型`int`的变量。Groovy会根据该实例的使用方式来决定将其存储为`int`类型还是`Integer`类型。Groovy在对自动装箱的处理上要比Java略胜一筹。在Java中，自动装箱和自动拆箱会涉及类型之间的转换。而另一方面，Groovy就简单地将其当作对象，所以不需要反复地转换类型。

在2.0版本之前，Groovy中所有基本类型都被看作对象。为了改进性能，也为了能在基本类型的操作上使用更为直接的字节码，从2.0版本起，Groovy做了一些优化。基本类型只在必要时才会被看作对象，比如，在其上调用了方法，或者将其传给了对象引用。否则，Groovy会在字节码级别将其保留为基本类型。

### 2.9.2 `for-each`

Groovy对循环的支持优于Java（参见2.1.2节）。在Groovy中仍然可以使用传统的`for`循环（也就是`for(int i = 0; i < 10; i++) {...}`）。或者，如果喜欢Java 5支持的更简单的循环形式，也可以使用。在Java 5中，实现了`Iterable`接口的对象可以用于`for-each`循环中，如下面的例子所示：

**GroovyForJavaEyes/ForEach.java**

```
// Java代码
String[] greetings = {"Hello", "Hi", "Howdy"};

for(String greet : greetings) {
  System.out.println(greet);
}
```

Groovy中可以像下面这样重写该例子：

**GroovyForJavaEyes/ForEach.groovy**

```
String[] greetings = ["Hello", "Hi", "Howdy"]
for(String greet : greetings) {
  println greet
}
```

在Java风格的`for-each`循环中，Groovy要求指明类型（即前面例子中的`String`），或者使用`def`。如果不想指定类型，则要使用`in`关键字代替冒号（:），如下面例子所示：

**GroovyForJavaEyes/ForEach.groovy**

```
for(greet in greetings) {
  println greet
}
```

相对于Java风格的`for-each`语法，在Groovy中我们更喜欢使用带有`in`的`for`语句。作为一种选择，还可以使用`each()`这一内部迭代器（参见第6章）。

### 2.9.3　enum

Groovy提供了对`enum`的支持，这是Java 5为解决枚举问题而引入的特性。它是类型安全的（比如，我们可以区分得出用`enum`表示的衬衫尺寸和一周中的每一天），还具有可打印、可序列化等特点。

下面例子定义了我们可以购买的咖啡饮品的容量规格：

**GroovyForJavaEyes/UsingCoffeeSize.groovy**

```
enum CoffeeSize { SHORT, SMALL, MEDIUM, LARGE, MUG }
def orderCoffee(size) {
  print "Coffee order received for size $size: "
  switch(size) {
    case [CoffeeSize.SHORT, CoffeeSize.SMALL]:
      println "you're health conscious"
      break
    case CoffeeSize.MEDIUM..CoffeeSize.LARGE:
      println "you gotta be a programmer"
      break
    case CoffeeSize.MUG:
      println "you should try Caffeine IV"
      break
  }
}
orderCoffee(CoffeeSize.SMALL)
orderCoffee(CoffeeSize.LARGE)
```

```
orderCoffee(CoffeeSize.MUG)
print 'Available sizes are: '
for(size in CoffeeSize.values()) {
    print "$size "
}
```



在前面的代码中，利用switch语句和enum上的迭代，很方便地产生了如下输出：

```
Coffee order received for size SMALL: you're health conscious
Coffee order received for size LARGE: you gotta be a programmer
Coffee order received for size MUG: you should try Caffeine IV
Available sizes are: SHORT SMALL MEDIUM LARGE MUG
```

可以在case语句中使用枚举值。特别地，我们可以使用一个值、一组值的列表或者值的一个区间。前面的代码囊括了上述这些用法。

Groovy也支持为Java 5的enum定义构造器和方法。请看下面例子：

**GroovyForJavaEyes/AgileMethodologies.groovy**

```
enum Methodologies {
    Evo(5),
    XP(21),
    Scrum(30);

    final int daysInIteration
    Methodologies(days) { daysInIteration = days }

    def iterationDetails() {
        println "${this} recommends $daysInIteration days for iteration"
    }
}

for(methodology in Methodologies.values()) {
    methodology.iterationDetails()
}
```

看一下在这个enum上迭代产生的输出：

```
Evo recommends 5 days for iteration
XP recommends 21 days for iteration
Scrum recommends 30 days for iteration
```

## 2.9.4　变长参数

利用Java 5的变长参数特性，可以向方法（比如printf()）传递数目不等的参数。要在Java中使用这一特性，我们使用一个省略符号（...）标记方法末尾的形参，比如public static Object max(Object... args)。这是语法糖，Java在调用时会把所有实参放入一个数组中。

Groovy以两种方式支持Java 5的变长参数特性，除了支持使用省略符号标记形参，对于以数

组作为末尾形参的方法，也可以向其传递数目不等的参数。

下面例子演示了Groovy支持的这两种方式：

**GroovyForJavaEyes/VarArgs.groovy**

```
def receiveVarArgs(int a, int... b) {
  println "You passed $a and $b"
}

def receiveArray(int a, int[] b) {
  println "You passed $a and $b"
}

receiveVarArgs(1, 2, 3, 4, 5)
receiveArray(1, 2, 3, 4, 5)
```

从输出可以看到，这两个版本都接受数目可变的实参：

```
You passed 1 and [2, 3, 4, 5]
You passed 1 and [2, 3, 4, 5]
```

对于接受变长参数或者以数组作为末尾形参的方法，可以向其发送数组或离散的值，Groovy知道该做什么。

在发送数组而非离散值时，请务必谨慎。Groovy会将包围在方括号中的值看作`ArrayList`的一个实例，而不是纯数组。所以如果简单地发送如`[2, 3, 4, 5]`这样的值，将出现`MethodMissingException`。要发送数组，可以定义一个指向该数组的引用，或使用`as`操作符。

**GroovyForJavaEyes/VarArgs.groovy**

```
int[] values = [2, 3, 4, 5]
receiveVarArgs(1, values)
receiveVarArgs(1, [2, 3, 4, 5] as int[])
```

大多数情况下，Groovy把类型看作可选的，但是这里我们看到，指定类型可以改变语义。

## 2.9.5　注解

Java中可以使用注解来表示元数据，而且Java 5带来了一些预定义的注解，比如`@Override`、`@Deprecated`和`@SuppressWarnings`。

Groovy中也可以定义和使用注解，而且定义注解的语法与Java相同。

在使用框架时经常会用到注解，比如，JUnit 4.0使用了`@Test`注解。如果正在使用像Hibernate、JPA、Seam和Spring这样的框架，我们就会发现Groovy对注解的支持非常有用。

对于Java中与编译相关的注解，Groovy的处理方式有所不同。例如，`groovyc`会忽略`@Override`。

### 2.9.6 静态导入

在Java中，静态导入可以帮助我们把一个类的静态方法导入到我们的命名空间中，所以无需指定类名即可引用它们。例如，如果将下面的语句放到Java代码中：

```
import static Math.random;
```

那么就可以像下面这样调用，而非使用Math.random()：

```
double val = random();
```

Java中的静态导入改进了工作的安全性。如果我们定义了几个静态导入，或者使用*导入了一个类的所有静态方法，无疑可以让那些想知道这些方法从何而来的程序员感觉莫名其妙。Groovy以两种形式扩展了这一优势。首先，它实现了静态导入。我们可以像在Java中那样使用。当然可以随意丢掉分号，它们在Groovy中是可选的。其次，在Groovy中可以为静态方法和类名定义别名。要定义别名，需要在import语句中使用as操作符：

```
import static Math.random as rand
import groovy.lang.ExpandoMetaClass as EMC

double value = rand()
def metaClass = new EMC(Integer)
assert metaClass.getClass().name == 'groovy.lang.ExpandoMetaClass'
```

这里为Math.random()方法创建了别名rand()，也为ExpandoMetaClass创建了别名EMC。现在可以分别使用rand()和EMC来代替Math.random()和ExpandoMetaClass了。

### 2.9.7 泛型

Groovy是支持可选类型的动态类型语言，作为Java的超集，它也支持泛型。然而，Groovy编译器不会像Java编译器那样执行类型检查（参见2.11.2节），不要期望Groovy编译器会像Java编译器那样一开始就拒绝违规的代码。如有可能，这里Groovy的动态类型特性将与泛型类型相互作用，使我们的代码运行起来。为了解这两种编译器的明显差别，在下面的例子中，我们将向一个保存Integer的ArrayList中添加一些类型不同的值。

从Java代码开始：

**GroovyForJavaEyes/Generics.java**

```
// Java代码
import java.util.ArrayList;

public class Generics {
  public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(1);
```

```
    list.add(2.0);
    list.add("hello");

    System.out.println("List populated");
    for(int element : list) { System.out.println(element); }
  }
}
```

当使用Java编译器编译前面的Java代码时，遇到了编译错误：

```
Generics.java:8: error: no suitable method found for add(double)
    list.add(2.0);
        ^
    method ArrayList.add(int,Integer) is not applicable
      (actual and formal argument lists differ in length)
    method ArrayList.add(Integer) is not applicable
      (actual argument double cannot be converted to Integer
         by method invocation conversion)
Generics.java:9: error: no suitable method found for add(String)
    list.add("hello");
        ^
    method ArrayList.add(int,Integer) is not applicable
      (actual and formal argument lists differ in length)
    method ArrayList.add(Integer) is not applicable
      (actual argument String cannot be converted to Integer
         by method invocation conversion)
2 errors
```

因为我们指定了`ArrayList`只保存`Integer`，对`add()`方法而言，除了`Integer`和`int`（`int`会被自动装箱为`Integer`），Java编译器不接受其他任何类型的数据。

来看一下Groovy是如何处理的。将前面的代码复制到一个命名为Generics.groovy的文件中，然后运行`groovy Generics`。Groovy不会阻止我们执行这段代码：

```
List populated
1
2
Caught: org.codehaus.groovy.runtime.typehandling.GroovyCastException:
Cannot cast object 'hello' with class 'java.lang.String' to class 'int'
org.codehaus.groovy.runtime.typehandling.GroovyCastException:
Cannot cast object 'hello' with class 'java.lang.String' to class 'int'
        at Generics.main(Generics.java:12)
        at Generics.invokeMethod(Generics.java)
        at RunGenerics.run(RunGenerics.groovy:1)
```

在调用`add()`方法的过程中，Groovy更大程度上是将类型信息看作一个建议。当对集合进行循环时，Groovy会尝试将其中的元素强制转换为`int`。如果无法转换，则会导致运行时错误。

Groovy在支持动态行为的同时支持泛型。前面的代码示例也说明了这两种概念有趣的相互作用。对于Groovy的这种双重性，我们一开始可能会感到惊讶，但是当学到Groovy元编程（参见第三部分）的好处时，你会看到其意义。

泛型的用处在Groovy中并没有完全丧失。如果我们愿意对元编程功能做一些折中，Groovy 2.*x* 也在部分代码上提供了严格的类型检查，具体参见3.8节。

## 2.10　使用 Groovy 代码生成变换



语言设计者往往要面对这样的问题，一方面想让语言演进，而另一方面又不愿意修改语法，因为这会影响性能、复杂性和语义正确性。Groovy巧妙地缓解了这种紧张。Groovy并没有修改语言的核心语法，其编译器会识别选定的注解并生成相应代码。本节将介绍一些这样的注解。第16章将介绍如何为定制的变换创建自己的注解。

Groovy在`groovy.transform`包和其他一些包中提供了很多代码生成注解。本节将探讨其中的一部分。

### 2.10.1　使用@Canonical

如果要编写的`toString()`方法只是简单地显示以逗号分隔的字段值，则可以使用`@Canonical`变换让Grooovy编译器帮来干这个活。默认情况下，它生成的代码会包含所有字段。不过可以让它仅包含特定字段，而去掉其他字段，比如下面这个例子：

**GroovyForJavaEyes/Annotations.groovy**

```
import groovy.transform.*

@Canonical(excludes="lastName, age")
class Person {
  String firstName
  String lastName
  int age
}

def sara = new Person(firstName: "Sara", lastName: "Walker", age: 49)
println sara
```

Groovy排除了我们提到的字段，打印了类名，类名后面是剩余字段的值，在输出中会看到：

```
Person(Sara)
```

### 2.10.2　使用@Delegate

只有当派生类是真正可替换的，而且可代替基类使用时，继承才显示出其优势。从纯粹的代码复用角度看，对于其他大部分用途，委托要优于继承。然而在Java中我们不太愿意使用委托，因为会导致代码冗余，而且需要更多工作。Groovy使委托变得非常容易，所以我们可以做出正确的设计选择。

要更好地理解委托，我们从一个Worker类入手，该类中有几个方法。Expert类有一个与Worker类的名字和签名都相同的方法。不出所料，Manager类什么都不干。但是它擅长把工作委托给别人，所以其中的两个字段使用@Delegate注解标记了。

**GroovyForJavaEyes/Annotations.groovy**

```
class Worker {
  def work() { println 'get work done' }
  def analyze() { println 'analyze...' }
  def writeReport() { println 'get report written' }
}
class Expert {
  def analyze() { println "expert analysis..." }
}
class Manager {
  @Delegate Expert expert = new Expert()
  @Delegate Worker worker = new Worker()
}
def bernie = new Manager()
bernie.analyze()
bernie.work()
bernie.writeReport()
```

在编译时，Groovy会检查Manager类，如果该类中没有被委托类中的方法，就把这些方法从被委托类中引入进来。因此，首先它会引入Expert类中的analyze()方法。而从Worker类中，只会把work()和writeReport()方法因为进来。这时候，因为从Expert类带来的analyze()方法已经出现在Manager类中，所以Worker类中的analyze()方法会被忽略。

对于引入的每个方法，Groovy会简单地把对该方法的调用路由给实例上的相应方法，就像这样：public Object analyze() { expert.analyze() }。委托类会对新获得的方法做出响应，在下面的输出中可以看到：

```
expert analysis...
get work done
get report written
```

因为有了@Delegate注解，Manager类是可扩展的。如果在Worker或Expert类上添加或去掉了方法，不必对Manager类做任何修改，相应的变化就会生效。只需要重新编译代码，剩下的事Groovy会处理。

## 2.10.3　使用@Immutable

不可变对象天生是线程安全的，将其字段标记为final是很好的实践选择。如果用@Immutable注解标记一个类，Groovy会将其字段标记为final的，并且额外为我们创建一些便捷方法，从而使得“做正确的事情”变得更容易了。

下面在CreditCard类中使用一下这个注解。

GroovyForJavaEyes/Annotations.groovy

```
@Immutable
class CreditCard {
  String cardNumber
  int creditLimit
}

println new CreditCard("4000-1111-2222-3333", 1000)
```

作为反馈，Groovy给我们提供了一个构造器，其参数以类中字段定义的顺序依次列出。在构造时间过后，字段就无法修改了。此外，Groovy还添加了hashCode()、equals()和toString()方法。运行所提供的构造器和toString()方法，看一下输出：

```
CreditCard(4000-1111-2222-3333, 1000)
```

可以使用@Immutable注解轻松地创建轻量级的不可变值对象。在基于Actor模型的并发应用中，线程安全是个大问题，而这些不可变值对象是作为消息传递的理想实例。

## 2.10.4　使用@Lazy

我们想把耗时对象的构建推迟到真正需要时。完全可以懒惰与高效并得，编写更少的代码，同时又能获得惰性初始化的所有好处。

下面的例子将推迟创建Heavy实例，直到真正需要它时。既可以在声明的地方直接初始化实例，也可以将创建逻辑包在一个闭包中。

GroovyForJavaEyes/Annotations.groovy

```
class Heavy {
  def size = 10
  Heavy() { println "Creating Heavy with $size" }
}

class AsNeeded {
  def value

  @Lazy Heavy heavy1 = new Heavy()
  @Lazy Heavy heavy2 = { new Heavy(size: value) }()

  AsNeeded() { println  "Created AsNeeded" }
}

def asNeeded = new AsNeeded(value: 1000)
println asNeeded.heavy1.size
println asNeeded.heavy1.size
println asNeeded.heavy2.size
```

Groovy不仅推迟了创建，还将字段标记为volatile，并确保创建期间是线程安全的。实例会在第一次访问这些字段的时候被创建，在输出中可以看到：

```
Created AsNeeded
Creating Heavy with 10
10
10
Creating Heavy with 10
1000
```

另一个好处是，@Lazy注解提供了一种轻松实现线程安全的虚拟代理模式（virtual proxy pattern）的方式。

### 2.10.5　使用@Newify

在Groovy中，经常会按照传统的Java语法，使用new来创建实例。然而，在创建DSL时，去掉这个关键字，表达会更流畅。@Newify注解可以帮助创建类似Ruby的构造器，在这里，new是该类的一个方法。该注解还可以用来创建类似Python的构造器（也类似Scala的applicator），这里可以完全去掉new。要创建类似Python的构造器，必须向@Newify注解指明类型列表。只有将auto=false这个值作为一个参数设置给@Newify，Groovy才会创建Ruby风格的构造器。

可以在不同的作用域中使用@Newify注解，比如类或方法，如下面例子所示：

**GroovyForJavaEyes/Annotations.groovy**

```
@Newify([Person, CreditCard])
def fluentCreate() {
  println Person.new(firstName: "John", lastName: "Doe", age: 20)
  println Person(firstName: "John", lastName: "Doe", age: 20)
  println CreditCard("1234-5678-1234-5678", 2000)
}

fluentCreate()
```

输出表明，借助该注解，可以使用Ruby和Python风格创建实例。

```
Person(John)
Person(John)
CreditCard(1234-5678-1234-5678, 2000)
```

在创建DSL时，@Newify注解非常有用，它可以使得实例创建更像是一个隐式操作。

### 2.10.6　使用@Singleton

要实现单件模式，正常来讲，我们会创建一个静态字段，并创建一个静态方法来初始化该字段，然后返回单件实例。我们必须确保该方法是线程安全的，同时还要决定是否要惰性创建该单

件。而通过使用@Singleton变换则完全可以避免这种麻烦，如下面例子所示：

**GroovyForJavaEyes/Annotations.groovy**

```
@Singleton(lazy = true)
class TheUnique {
  private TheUnique() { println 'Instance created' }

  def hello() { println 'hello' }
}
println "Accessing TheUnique"
TheUnique.instance.hello()
TheUnique.instance.hello()
```

当运行这段代码时，实例会在第一次调用instance属性时被创建，该属性会映射到getInstance()方法。

```
Accessing TheUnique
Instance created
hello
hello
```

这里使用@Singleton注解标记了TheUnique类，以生成静态的getInstance()方法。因为此处将lazy属性的值设为了true，所以会将实例的创建延迟到请求时。可以检查一下生成的代码：把前面的代码复制粘贴到groovyConsole中，然后选择Script菜单下的Inspect AST菜单项。

```
public class TheUnique implements
  groovy.lang.GroovyObject extends java.lang.Object {

    private static volatile TheUnique instance
    //...

    private TheUnique() {
        metaClass = /*BytecodeExpression*/
        this.println('Instance created')
    }

    public java.lang.Object hello() {
        return this.println('hello')
    }

    public static TheUnique getInstance() {
        if ( instance != null) {
            return instance
        } else {
            synchronized (TheUnique) {
                if ( instance != null) {
                    return instance
                } else {
```

```
                return instance = new TheUnique()
            }
        }
    }
}
//...
```

Groovy不仅将实例创建延迟到了最后责任时刻，还保证创建部分是线程安全的。

---

**警告**　使用@Singleton注解，会使目标类的构造器成为私有的，这在我们意料之中，不过因为Groovy实现并不区分公开还是私有，所以在Groovy内仍可使用new关键字来创建实例。但是，必须谨慎恰当地使用这个类，并留心代码分析工具和集成开发环境给出的警告。

---

除了前面提到的这些，Groovy还提供了一个便于使用的注解，用以解决在继承带有多个构造器的类时所需要做的苦差事。在Java中，即使我们几乎不想调用到相应父类的构造器，它还是会强制我们实现这多个构造器。如果使用@InheritConstructors注解该类，则Groovy会为我们生成这些构造器。

以上是Groovy优秀的一面，但是作为客观的程序员，我们也必须承认，有些东西可能会绊我们一下。下一节就来介绍一些陷阱，帮助我们在需要时保持警惕。

## 2.11　陷阱

阅读本书过程中，我们将看到Groovy的很多不错的功能，但是在使用Groovy时也确实存在一些“陷阱”——从小小的烦恼到可能令你吃惊的问题。接下来，我们将探讨一些陷阱。

### 2.11.1　Groovy 的==等价于 Java 的 equals()

在Java中，==和equals()是一个混乱之源，而Groovy加剧了这种混乱。Groovy将==操作符映射到了Java中的equals()方法。假如我们想比较引用是否相等（也就是原始的==的语义），该怎么办呢？必须使用Groovy中的is()。下面通过一个例子来理解其区别。

**GroovyForJavaEyes/Equals.groovy**

```
str1 = 'hello'
str2 = str1
str3 = new String('hello')
str4 = 'Hello'

println "str1 == str2: ${str1 == str2}"
println "str1 == str3: ${str1 == str3}"
println "str1 == str4: ${str1 == str4}"
```

```
println "str1.is(str2): ${str1.is(str2)}"
println "str1.is(str3): ${str1.is(str3)}"
println "str1.is(str4): ${str1.is(str4)}"
```

来看一下Groovy中==操作符的行为以及使用is()方法的结果：

```
str1 == str2: true
str1 == str3: true
str1 == str4: false
str1.is(str2): true
str1.is(str3): false
str1.is(str4): false
```

观察发现，Groovy的==映射到equals()，这个结论并不总是成立，当且仅当该类没有实现Comparable接口时，才会这样映射。如果实现了Comparable接口，则==会被映射到该类的compareTo()方法。

下面例子说明了这种行为。

**GroovyForJavaEyes/WhatsEquals.groovy**

```
class A {
  boolean equals(other) {
    println "equals called"
    false
  }
}

class B implements Comparable {
  boolean equals(other) {
    println "equals called"
    false
  }

  int compareTo(other) {
    println "compareTo called"
    0
  }
}

new A() == new A()
new B() == new B()
```

下面的输出显示，Comparable的优先级高：

```
equals called
compareTo called
```

通过输出可以看到，在实现了Comparable接口的类上，==操作符选择了compareTo()，而不是equals()。

**注意**　在比较对象时，请首先问一下自己，要比较的是引用还是值。然后再问一下，是不是使用了正确的操作符。

### 2.11.2　编译时类型检查默认为关闭

Groovy的类型是可选的。然而，Groovy编译器groovyc大多数情况下不会执行完整的类型检查（第3章会介绍Groovy如何支持选择性的类型检查），而是在遇到类型定义时执行强制类型转换。如果要把一个字符串赋给一个Integer类型的变量：

**GroovyForJavaEyes/NoTypeCheck.groovy**

```
Integer val = 4
val = 'hello'
```

这段代码可以正常编译，没有错误。但当尝试运行编译生成的Java字节码时，会出现GroovyCastException异常，在输出中可以看到：

```
org.codehaus.groovy.runtime.typehandling.GroovyCastException:
Cannot cast object 'hello' with class 'java.lang.String'
to class 'java.lang.Integer'
```

Groovy编译器不会验证类型，相反，它只是进行强制类型转换，然后将其留给运行时处理。这可以通过分析生成的字节码来验证（使用javap -c ClassFileName命令可以一窥可读的字节码形式）：

```
...
35: ldc #71 // String hello
37: astore_3
38: aload_3
39: ldc #65 // class java/lang/Integer
41: invokestatic  #75 // Method ...castToType:(...)...
44: checkcast #65 // class java/lang/Integer
...
```

所以在Groovy中，x = y在语义上等价于x = (ClassOfX)(y)。类似地，如果调用了一个不存在的方法（比如下面例子中调用了不存在的blah方法），也不会出现编译错误：

**GroovyForJavaEyes/NoTypeCheck.groovy**

```
Integer val = 4
val.blah()
```

不过运行时会出现MissingMethodException异常：

```
groovy.lang.MissingMethodException:
No signature of method: java.lang.Integer.blah() is applicable
for argument types: () values: []
Possible solutions: each(groovy.lang.Closure), with(groovy.lang.Closure),
```

```
plus(java.lang.Character), plus(java.lang.String), plus(java.lang.Number),
wait()
```

在第13章你将看到,这实际上是个优点。我们可以在代码编译时和执行时动态注入缺失的方法。

Groovy编译器可能看上去不够严格,但是对于Groovy的动态和元编程等强项而言,这种行为是必要的[①]。在2.*x*版本中,我们可以关闭这种动态类型特性,并增强编译时类型检查,3.8节及3.8.1节将会介绍。



### 2.11.3 小心新的关键字

`def`和`in`都是Groovy中的新关键字。`def`用于定义方法、属性和局部变量。`in`用于在`for`循环中指定循环的区间,比如`for(i in 1..10)`。

将这些关键字用作变量名或方法名可能会带来问题,尤其是当把现有的Java代码当作Groovy代码时。

定义名为`it`的变量也是不明智的。尽管Groovy不会抱怨什么,但是如果在闭包内使用了这样的变量,它引用的是闭包的参数,而不是类中的一个字段——隐藏变量可无助于偿还技术债[②]。

### 2.11.4 别用这样的代码块

下面是合法的Java代码:

**GroovyForJavaEyes/Block.java**

```
// Java代码
public void method() {
  System.out.println("in method1");

  {
    System.out.println("in block");
  }
}
```

Java中的代码块定义了一个新的作用域,但是Groovy会感到困扰。Groovy编译器会错误地认为我们是要定义一个闭包,并给出编译错误。在Groovy中,方法内不能有任何这样的代码块。

### 2.11.5 闭包与匿名内部类的冲突

Groovy的闭包是使用花括号(`{...}`)定义的,而定义匿名内部类也是使用花括号。如下面例子所示,当构造器接收一个闭包作为参数时,就出现问题了:

---

① http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic
② http://martinfowler.com/bliki/TechnicalDebt.html

**GroovyForJavaEyes/Calibrator.groovy**

```
class Calibrator {
  Calibrator(calculationBlock) {
    print "using..."
    calculationBlock()
  }
}
```

正常情况下，可以通过把一个代码块附到函数调用末尾，将闭包传递给函数：`instance.method() {...}`。按照这种习惯，我们可以通过向`Calibrartor`的构造器传递一个闭包来实例化一个实例，如下面代码所示：

**GroovyForJavaEyes/AnonymousConflict.groovy**

```
def calibrator = new Calibrator() {
  println "the calculation provided"
}
```

在这个例子中，我们是要调用`Calibrator`类的构造器，它接受一个闭包作为参数。事与愿违，Groovy却认为我们是要创建一个匿名内部类，因而报告了一个错误。

```
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
.../code/GroovyForJavaEyes/AnonymousConflict.groovy:
2: unexpected token: println @ line 2, column 3.
     println "the calculation provided"
     ^

1 error
```

要绕开这个陷阱，必须修改调用方式，将闭包放在构造器调用语句的圆括号内。我们仍然可以在调用时定义闭包，或传递一个引用该闭包的变量。

**GroovyForJavaEyes/AnonymousConflictResolved.groovy**

```
def calibrator1 = new Calibrator({
  println "the calculation provided"
})
def calculation = { println "another calculation provided" }
def calibrator2 = new Calibrator(calculation)
```

运行这段代码，验证这个版本没有让Groovy迷惑，并且达到了将闭包传递给构造器的预期结果。

```
using...the calculation provided
using...another calculation provided
```

这只是个小麻烦；与传递内联的闭包相比，传递引用给闭包噪音会小一些。

## 2.11.6 分号总是可选的

使用从C语言派生而来的语言的程序员，小拇指可受了不少年的罪，在Groovy中他们可以轻松一下了。因为不必在语句末尾放一个分号（;）。丢掉分号是有好处的，这有助于创建领域特定语言（DSL）。尽管分号是可选的，但是在一行中放置多条语句时，分号还是有用的。至少有一个地方，分号是必不可少的，如下面例子所示：



**GroovyForJavaEyes/SemiColon.groovy**

```
class Semi {
  def val = 3

  {
    println "Instance Initializer called..."
  }
}

println new Semi()
```

我们本打算让这个代码块成为类的一个实例初始化器，但是Groovy迷惑了，它把实例初始化器看成了一个闭包，并给出了如下错误：

```
Caught: groovy.lang.MissingMethodException:
No signature of method: java.lang.Integer.call()
is applicable for argument types: (Semi$_closure1)
values: {Semi$_closure1@be513c}
  at Semi.<init>(SemiColon.groovy:3)
  at SemiColon.run(SemiColon.groovy:10)
  at SemiColon.main(SemiColon.groovy)
```

如果把`def val = 3`改写为`def val = 3;`，代码即可正常运行。现在Groovy把这个代码块识别成了实例初始化器，而不是附加到属性定义上的一个部分。

如果要使用的是一个静态初始化器，而不是实例初始化器，那就没有这个问题了。如果有理由同时使用这两种初始化器，可以将静态初始化器放在实例初始化器之前，从而避免使用分号。

## 2.11.7 创建基本类型数组的不同语法

要在Groovy中创建基本类型的数组，不能使用我们在Java中所习惯的符号。

在Java中，可以像下面这样创建整型的数组：

**GroovyForJavaEyes/ArrayInJava.java**

```
int[] arr = new int[] {1, 2, 3, 4, 5};
```

而在Groovy中，上述代码会导致编译错误。Groovy以如下方式定义基本类型的数组：

GroovyForJavaEyes/ArrayInGroovy.groovy

```
int[] arr = [1, 2, 3, 4, 5]

println arr
println "class is " + arr.getClass().name
```

输出表明，所创建实例的类型为`[I`，JVM用它表示`int[]`。

```
[1, 2, 3, 4, 5]
class is [I
```

如果省略掉左侧的类型信息`int[]`，Groovy会假设我们在创建`ArrayList`的实例（参见2.9.4节），所以在这个例子中指定类型非常关键。作为一种选择，还可以使用`as`操作符来创建数组：

```
def arr = [1, 2, 3, 4, 5] as int[]
println arr2
println "class is " + arr2.getClass().name
```

Groovy使得创建`ArrayList`类型的实例更为容易了，但是要创建数组，则必须付出更多努力。

以上是一些在Groovy中编程时可能遇到的陷阱。因为我们有Java背景，所以可以通过了解Groovy与Java的不同来获益。http://groovy.codehaus.org/Differences+from+Java列出了Groovy与Java的差别，很不错。

本章介绍了很多东西。你现在知道了在Groovy中如何编写类，学到了一些Groovy的习惯用法，也了解了一些Groovy编写代码的方式。你还了解到，必要的情况下我们可以回到Java语法。现在，你就可以开始实验和把玩Groovy了，不必等到学完这本书。然而我们还有很多东西要学。术语动态类型和可选类型已经出现过多次，所以下一章将介绍这些主题，并探讨如何在Groovy中利用它们。

# 第3章 动态类型



动态类型语言中的类型是在运行时推断的，方法及其实参也是在运行时检查的。通过这种能力，可以在运行时向类中注入行为，从而使代码比严格的静态类型具有更好的可扩展性。

本章介绍动态类型的优点，以及如何在Groovy中使用动态类型。借助动态类型，可以用比Java更少的代码创建灵活的设计。将实参的类型验证推迟到运行时这一特性为Groovy中的多态注入了活力。利用*多方法*（multimethods）这一工具，可以为与实参的运行时类型相关的操作提供替换行为。本章还将介绍如何使用Groovy中的静态编译选项，妥善地利用这些强大的特性。

## 3.1 Java 中的类型

编译时类型检查所提供的“安全性”让人产生了依赖心理。但是类型安全中的安全性和社会保障中的保障同样让人“放心”。

假设Car类有两个属性，一个是year，一个是一个Engine类的实例。现在要实现对Car类对象的复制。先忽略Java中的深度复制问题[①]。为了提供复制功能，需要实现Cloneable接口，并提供一个public的clone()方法。Object类的clone()方法可以创建对象的一个浅副本（shallow copy）。不过这里希望不同的Car实例所包含的Engine也不同。因此，在使用clone()这个基本的方法实现复制时，还需要稍作修改，使其拥有自己的Engine，如下面代码所示：

**TypesAndTyping/Car.java**

```
//Java代码
public Object clone() {
  try {
    Car cloned = (Car) super.clone();
    cloned.engine = (Engine) engine.clone();
  return cloned;
} catch(CloneNotSupportedException ex) {
  return null; // 不会发生这种情况，这是为了取悦编译器
}
```

① 参见我的文章“Why Copying an Object Is a Terrible Thing to Do”（为什么复制对象很可怕），其中讨论了在Java中复制对象的问题：http://www.agiledeveloper.com/articles/cloning072002.htm。

这段代码噪音很大。首先，编译器坚持让我们处理`CloneNotSupportedException`，而且就在实现复制的方法中处理。其次，当在`Car`类的实例方法中调用`super.clone()`时，其实已经能够明确目标是另一个`Car`。然而编译器还是固执地要求对调用结果进行强制类型转换。下一条复制`Engine`的语句也是如此。此外，当准备好在一个`Car`实例上调用`clone()`方法时，还需要再次强制类型转换，以便把调用的结果保存到一个`Car`引用中。有时候静态类型检查只会带来烦恼，而且还会降低效率。好的类型检查应该像一个好政府——只做必要的事，而不能阻碍发展。然而，在大部分时间内，Java编译器都是一个障碍。

编译时类型检查自有其价值。不过如今的集成开发环境（IDE）让开发代码和运行测试变得非常容易，所以往往让IDE来保存相关的编辑文件，并在必要时编译代码。当尝试运行测试失败时，再来解决问题。因此，在重复快速的“编辑–运行–测试”循环的同时，不必再对编译错误、运行时错误和测试失败作太多区分。关键在于让代码持续工作并让所有测试通过。

## 3.2　动态类型

动态类型放宽了对类型的要求，使语言能够根据上下文判定类型。

动态类型有什么优点？放弃在编译时或代码编辑时对类型进行验证或确认的优势，这值得吗？动态类型有两大优点，使这种舍弃利大于弊。

首先，可以在不知道方法具体细节的情况下编写对象上的调用语句。在运行期间，对象会动态地响应方法或消息。在静态类型语言中，使用多态可在某种程度上实现这种动态行为。然而，大部分静态类型语言把继承和多态捆绑在了一起。它们强迫我们遵循某个结构，而不是遵循实际行为。真正的多态并不关注类型——把一个消息发送给一个对象，在运行时，它自会确定所要使用的相应实现。因此，动态类型语言可以实现比传统的静态类型语言更高程度的多态。

其次，不必用大量的强制类型转换操作来取悦编译器，就像3.1节中的例子那样。

一种聪明而且愿意配合程序员的语言，当然谁都愿意使用。工作效率会更为高效，在一定程度上，这都得益于少了很多繁文缛节。

在使用静态语言工作时，那感觉就像是有个唠叨的婆婆站在旁边，盯着我们的一举一动。对于将某些实现推迟到后面某个时间（代码执行前），静态类型语言也没有提供充分的灵活性。相反，使用动态类型语言工作时，就像有位和蔼的爷爷站在旁边，让我们做实验，把事情弄清楚，保持创造性，而他只是站在一旁，在必要时才提供协助。

第一个优点（真正实现了多态）极大地改进了设计应用的方式，3.4节还会详细讨论。

## 3.3 动态类型不等于弱类型

在静态类型语言中，要在编译时指定变量、引用等数据的类型，而且很多编译器会坚持要求这么做。比如在C/C++中，必须指定变量类型，要么是诸如`int`或`double`等基本类型，要么就是某个具体类的类型。然而，如果把该变量强制转换为一个错误类型，又会怎么样呢？编译器会阻止吗？并不会。运行时，程序的命运又当如何？那得看运气。如果幸运的话，程序会崩溃。如果不幸，可能会一直等到我们做重要的演示时才出现崩溃或错误行为。根据内存如何配置，调用是否多态，虚函数表如何组织[1]①等不同条件，最终表现也很难预测。如果仔细去听，可能还会听到编译器的嘲笑，嘲笑我们依赖它假装提供的类型安全。这是静态类型和运行时弱类型相结合的一个例子。

在下面的图中，我们基于静态对比动态、强类型对比弱类型对常见语言做了分类。

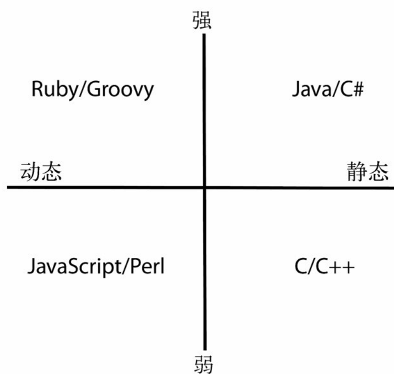



图3-1 所选语言的分类：静态对比动态、强类型对比弱类型

Java是一种静态类型语言，但它是强类型的。编译器会检查类型，但是如果我们强制转换成了某个错误类型，运行时会给我们抓出来。

动态类型语言，比如Groovy，不会在代码编辑时或编译时执行类型检查。然而，如果把一个对象当作错误的类型，Groovy会在运行时毫不含糊地提示出来。把实际的验证推迟到了运行时；我们得以在编码和编译时，以及在代码执行时修改程序的结构。JVM上的动态类型语言说明，动态类型并不意味着弱类型。

① 有些语言，比如C++，会维护一个方法分派表，其中保存了多态方法的地址，参见Margaret A. Ellis和Bjarne Stroustrup合著的*The Annotated C++ Reference Manual* [ES90]一书。

## 3.4　能力式设计

作为Java程序员，我们严重依赖接口。我们推崇“契约式设计”（Design By Contract），在这种设计中，接口定义了交流的契约，类负责实现并遵守这些契约——参见Bertrand Meyer的*Object-Oriented Software Construction*[Mey97]一书。

商业契约是个好东西，可以帮助确保特定的预期目标得以实现。然而，契约最好不要太过严格，要有一定的灵活性，以便以可接受的方式满足或超出预期。

软件契约也是类似的。基于接口的编程，尽管非常强大，但往往有很多限制。下面的例子突出了静态类型和动态类型之间的差别。

### 3.4.1　使用静态类型

假设需要搬一些重物，就会想找一个愿意帮忙、而且有能力的帮手。在Java中，代码可以写成下面这样：

**TypesAndTyping/TakeHelp.java**

```
public void takeHelp(Man man) {
  //...
  man.helpMoveThings();
  //...
}
```

因为是静态类型，而且参数为Man，所以不会理会附近的某位愿意帮忙、而且有能力的女人（Woman）。下面扩充这段代码，以便可以寻求男人或是女人的帮助，来创建一个Human抽象类，其中包含helpMoveThings()方法。Man和Woman都可以提供对该方法的实现：

**TypesAndTyping/Human.java**

```
// Java代码
public abstract class Human {
  public abstract void helpMoveThings();

  //...
}
```

下面是接受一个Human的帮助的代码：

**TypesAndTyping/TakeHelp.java**

```
public void takeHelp(Human human) {
  //...
  human.helpMoveThings();
  //...
}
```

好了，现在任何人都可以帮我们搬东西了。然而，如果我们是塞伦盖蒂平原上的流浪者，一头友好的大象或许能提供帮助，但我们却没法利用它的友善——因为依赖的是Human，大象不会遵从这个契约。又该加以扩充了，这次引入一个Helper接口，其中包含helpMoveThings()方法：

**TypesAndTyping/Helper.java**

```
// Java代码
public interface Helper {
    public void helpMoveThings();
}
```

然后Human、Elephant，只要可以提供帮助，都可以实现Helper。我们现在依赖Helper，可以接受来自实现了该接口的实例的帮助：



**TypesAndTyping/TakeHelp.java**

```
public void takeHelp(Helper helper) {
  //...
  helper.helpMoveThings();
  //...
}
```

至此，扩展需要付出很多努力。要使用多种多样的对象，意味着要创建接口，并修改依赖接口的代码。

### 3.4.2 使用动态类型

我们使用Groovy的动态类型能力再来看上节所举的那个“获得帮助”的例子：

**TypesAndTyping/TakeHelp.groovy**

```
def takeHelp(helper) {
  //...
  helper.helpMoveThings()
  //...
}
```

takeHelp()接受一个helper，但是没有指定其类型，这样类型默认为Object。此外，这里在它上面调用了helpMoveThings()方法。这就是能力式设计（Design By Capability）[①]。不同于让helper遵守某些显式的接口，我们利用了对象的能力——依赖一个隐式的接口。这被称作鸭子类型，它基于这一观点：“如果它走路像鸭子，叫起来也像鸭子，那它就是一只鸭子。”[②]

想要这种能力的类只需要实现该方法，而不需要扩展或实现任何东西。这样的结果就是少了

① 此方式与“契约式设计”相对应，书中多处对比这两种设计方式，因此参考“契约式设计”译为“能力式设计”。——译者注

② http://c2.com/cgi/wiki?DuckTyping

繁文缛节，增加了生产效率。如果机器有这种能力，可以帮我们搬东西，则不再需要修改代码，就能拿来使用。来看一些提供了我们想要的能力的类。

**TypesAndTyping/TakeHelp.groovy**

```
class Man {
  void helpMoveThings() {
    //...
    println "Man's helping"
  }
  //...
}

class Woman {
  void helpMoveThings() {
    //...
    println "Woman's helping"
  }
  //...
}

class Elephant {
  void helpMoveThings() {
    //...
    println "Elephant's helping"
  }
  void eatSugarcane() {
    //...
    println "I love sugarcanes..."
  }
  //...
}
```

下面是一个调用`takeHelp()`方法的例子：

**TypesAndTyping/TakeHelp.groovy**

```
takeHelp(new Man())
takeHelp(new Woman())
takeHelp(new Elephant())
```

再来看一下各自的效果：

```
Man's helping
Woman's helping
Elephant's helping
```

这些类没有扩展任何公共类，也没有实现任何公共接口，但是借助Groovy的动态特性，我们能够在`takeHelp()`方法中使用所有这些类。

因为来自Java背景，我们可能需要付出些努力才能习惯Groovy的动态特性，但是一旦感觉对了，就可以好好利用了。例如，在一个订单处理系统中，可以使用一个模拟（Mock）对象毫不费力地替换掉一个信用卡处理对象，以便进行快速的自动化测试，而不必提前做出优雅的设计决策。这也意味着，可以比较方便地加入一些设计反思，这为创建容易扩展的代码提供了更多的灵活性和动力。

### 3.4.3 使用动态类型需要自律



迄今为止，见识到利用动态类型能使代码多么简单、优雅和灵活了吧？但这其中是否存在风险呢？

- 在创建一个helper时，可能会敲错方法的名字。
- 没有类型信息，怎么知道给方法发什么呢？
- 如果把方法发给一个不能提供帮助的事物（一个不能搬动物体的对象），又会怎么样呢？

这些担心都是合理的，但是不必恐慌。这一节，我们来看看解决之道。

在编写代码时经常会出现拼写错误。大脑也经常会愚弄人：往往看到的其实是想看到的，而不是真正摆在那里的东西。因此，必须确保方法名的大小写正确，接受的形式参数正确。静态类型语言中的编译器会代为检查。而在动态类型语言中，要么不让编译器做，要么就是编译器本来就不做。这就需要依赖单元测试（参见18.2节）来确保其正确性。仅为这一目的编写单元测试相当古怪。不过，编译器生成的字节码也不一定就完全正确。仍然需要验证代码符合预期目标，不仅要符合输入的代码，还要符合我们的真实意图。

我在编程时是很依赖单元测试的，甚至在静态类型语言中也是如此。缺乏编译器支持（或是缺乏某个编译器）来验证这些内容，并没有令我苦恼。单元测试是很好的方法，而且动态类型需要开发者自律地做单元测试。要知道，使用动态类型语言编程，却没有使用单元测试的自律，就像是在玩火。

在一定程度上，类型可以帮助我们确定需要给一个方法发送什么对象或什么值。但这只是一个方面。在实践中，知道必须给一个方法发送一个`double`类型的值一般是不够的（除非我们想因毁掉了卫星而出名）[1]。自律的单元测试和良好的命名约定可以起到帮助。

最后，再来看一下是否一致的问题：如果发来的是一个不支持预期方法的对象，会怎么样？可以假定调用方负责保证所发送内容的合法性。如果发送了非法的对象，代码会失败，而且会抛给调用方一个异常。即使在编译的代码中，也必须处理前置条件破坏的问题，现在也一样，而且要求变得更多了。在特殊的情况下，如果想处理一些可供替换的或可选的行为，可以问一下对象，了解它是否能够完成我们预期的功能。Groovy的`respondsTo()`方法可以帮忙（参见11.2节）。假

---

① http://www.cnn.com/TECH/space/9909/30/mars.metric.02/

定有一个甘蔗农场，想分给帮助者一些甘蔗，但不是所有的帮助者都会吃甘蔗。我们可以问问帮助者是否喜欢：

**TypesAndTyping/TakeHelp.groovy**

```
def takeHelpAndReward(helper) {
  //...
  helper.helpMoveThings()

  if (helper.metaClass.respondsTo(helper, 'eatSugarcane'))
  {
    helper.eatSugarcane()
  }
  //...
}

takeHelpAndReward(new Man())
takeHelpAndReward(new Woman())
takeHelpAndReward(new Elephant())
```

我们询问了帮助者是不是可以接受甘蔗，如果可以，就分给它一些，如下面输出所示：

```
Man's helping
Woman's helping
Elephant's helping
I love sugarcanes...
```

如果使用时加以自律，能力式设计可以创建高度可扩展的、简洁的代码。代码中的强制类型转换和噪音会少一些，类层次结构也会更短。我们开始感觉编译器是在积极地辅助我们，而不是唱反调。

## 3.5　可选类型

Groovy是动态类型的，同时也是可选类型的。这意味着，既可以将类型的转盘拨到一个极端——不指定任何类型，让Groovy确定；也可以将它拨到另一个极端，精确地指定所要使用的变量或引用的类型。

请记住，Groovy是一门运行在JVM上的语言。可选类型有助于集成Groovy代码和Java的库、框架以及工具。有时候，Groovy的动态类型映射与当前使用的库、框架或工具并不匹配。这种情况在Groovy中并不突出——开发者可以轻松地切换类型模式，指明类型信息。可选类型在其他情况下也是有用的，比如有时候需要类型信息来生成数据库模式，或者创建GORM/Grails中的验证器。

试想，正在使用Groovy编写一个JUnit测试（参见18.2节）。在定义方法时，可以使用`def`关键字来说明返回类型为`Object`。因为JUnit要求测试方法为`void`，如果尝试运行使用`def`定义的测

试，就会遇到一个错误。相反，必须把方法定义为void，以满足JUnit。这里Groovy的可选类型就派上用场了。

回看一下图3-3，你可能会疑惑，如果Groovy是可选类型的，那为什么不放在静态类型和动态类型中间呢？这是因为，Groovy编译器（groovyc）不会进行完整的类型检查（具体细节参见2.11.2节）。如果我们写下X obj = 2，其中X是一个类，groovyc会简单地放一个强制类型转换操作，如X obj = (X) 2，让运行时动态确定这条语句是否合法。因此，即便Groovy允许使用类型，它仍然是动态类型的。



## 3.6　多方法

动态类型和动态类型语言改变了对象响应方法调用的方式。

Groovy支持多态，就像Java那样，但是它比基于目标对象的类型简单地分派方法走得更远。来看一下Java中的多态：

**TypesAndTyping/Employee.java**

```
// Java代码
public class Employee {
  public void raise(Number amount) {
    System.out.println("Employee got raise");
  }
}
```



Employee类中的raise()方法仅仅报告了一下它被调用了。现在看一下Executive类：

**TypesAndTyping/Executive.java**

```
// Java代码
public class Executive extends Employee {
  public void raise(Number amount) {
    System.out.println("Executive got raise");
  }

  public void raise(java.math.BigDecimal amount) {
    System.out.println("Executive got outlandish raise");
  }
}
```

Executive覆盖了Employee中的raise(Number amount)方法，它会报告Executive中的这个方法被调用了。同时提供了一个重载的版本——raise(java.math.BigDecimal amount)，它会打印获得了不同寻常的加薪。你预计下面的代码会调用哪个版本呢？

最后来看一下使用这些类的Java代码：

**TypesAndTyping/GiveRaiseJava.java**

```
// Java代码
import java.math.BigDecimal;
public class GiveRaiseJava {
  public static void giveRaise(Employee employee) {
    employee.raise(new BigDecimal(10000.00));
  }

  public static void main(String[] args) {
    giveRaise(new Employee());
    giveRaise(new Executive());
  }
}
```

代码中创建了一个`Employee`和一个`Executive`，调用了同样的`giveRaise()`方法，而该方法又会调用这些对象上的`raise()`方法。输出正如我们的预期：

```
Employee got raise
Executive got raise
```

`Employee`中的`raise()`方法是多态的，这意味着在运行时，被调用的方法依赖的并不是目标的引用类型，而是所引用对象的实际类型。不过这里有个限制。在运行时调用的方法必须接收`Number`作为一个参数，因为这是由基类`Employee`定义的。因此，编译器会把`BigDecimal`的实例看作`Number`。

这是一个标准的、日常使用的Java操作。没什么大不了的，对吧？但是当涉及Groovy的动态特性时，一切都不同了。Groovy深知Tony Hoare的那句名言："过早的优化是万恶之源。"[①]

当在Groovy中调用`raise()`方法时，它不会像Java中的代码那样按前面的顺序走。相反，形象地说，它会走到一个对象跟前，问一下："嘿，咱是不是有一个接收`java.math.BigDecimal`的`raise()`方法啊？"一个`Employee`对象会说，"没有，不过我可以接收一个`Number`。"而另一方面，一个`Executive`对象确实有一个接收`BigDecimal`的`raise()`方法，所以调用会被路由到这个对象。下面代码说明了这一行为，我们仍然使用前面Java定义的`Employee`类和`Executive`类，它们没有变化：

**TypesAndTyping/GiveRaise.groovy**

```
void giveRaise(Employee employee) {
  employee.raise(new BigDecimal(10000.00))
  // 和下面这条语句效果相同
  //employee.raise(10000.00)
}
```

① 这里的"过早的优化"指的是提前确定调用的版本，在`GiveRaiseJava.java`这个例子中，`raise(java.math.BigDecimal amount)`方法在重载解析时就被排除掉了，而Groovy则直到最后调用时才根据目标对象和所提供的参数确定实际要调用的方法版本。——译者注

```
giveRaise new Employee()
giveRaise new Executive()
```

Groovy报告的输出和Java不同：

```
Employee got raise
Executive got outlandish raise
```

如果一个类中有重载的方法，Groovy会聪明地选择正确的实现——不仅基于目标对象（调用方法的对象），还基于所提供的参数。因为方法分派基于多个实体——目标加参数，所以这被称作多分派或多方法（Multimethods）。

3

因为多方法机制，Groovy没有遭受Java的类型混淆问题之苦，感谢Neal Ford提供的这个Java示例。看一下下面使用了泛型的Java代码。`lst`引用的是一个`ArrayList<String>`实例，而`Collection<String>`类型的`col`引用的是同一实例。我们向`lst`中加入3个元素，然后移除1个。移除操作去掉了列表中的第一个元素。现在我们想调用`col.remove(0)`来移除另一个元素。然而，`Collection`接口的`remove()`方法想接收的是一个`Object`，所以Java把`0`装箱成一个`Integer`。因为这个`Integer`实例不是列表中的元素，所以这个方法调用没有移除掉任何东西。

**TypesAndTyping/UsingCollection.java**

```
//Java代码
import java.util.*;

public class UsingCollection {
  public static void main(String[] args) {
    ArrayList<String> lst = new ArrayList<String>();
    Collection<String> col = lst;

    lst.add("one");
    lst.add("two");
    lst.add("three");
    lst.remove(0);
    col.remove(0);

    System.out.println("Added three items, removed two, so 1 item to remain.");
    System.out.println("Number of elements is: " + lst.size());
    System.out.println("Number of elements is: " + col.size());
  }
}
```

输出显示出这段代码令人不爽的行为：

```
Added three items, removed two, so 1 item to remain.
Number of elements is: 2
Number of elements is: 2
```

再来看一下这段代码在Groovy中的表现。不需要对前面的代码做任何改动，只是简单地将其复制粘贴到一个名为`UsingCollection.groovy`文件中。然后运行`groovy UsingCollection`并

观察输出。可以看到，其输出与Java版本不同：

```
Added three items, removed two, so 1 item to remain.
Number of elements is: 1
Number of elements is: 1
```

Groovy的动态与多方法能力很好地处理了这种情况。在运行时，Groovy知道我们想去掉第一个元素，而不会招惹装箱操作这种不必要的麻烦，也就不会出现前面那种不正确的行为。

## 3.7　动态还是非动态

鉴于Groovy是一门支持可选类型的动态类型语言，那我们是应该指明类型，还是依赖动态类型呢？这方面其实没有真正的规则，但是我们当然可以养成一些偏好。

在使用Groovy编程时，我倾向于省略类型，不过我会为形参或变量选择表达性好的名字。这是因为，不指明类型，一来可以享受到鸭子类型的优点（参见3.4节），二来使应用模拟进行测试变简单了（参见18.2节）。

当然在必要的情况下，我也会选择指明类型。比如，JUnit要求测试方法为`void`类型，或者指明类型信息有很大的好处，比如在Grails对象关系映射（GORM）中把类型映射到数据库。

如果要为使用静态类型语言的程序员开发API，那我们会在静态类型的、面向客户的API中指明方法形参的类型。

从使用的角度看，社区倾向于总是指明方法签名中的类型。其优势是，在方法调用时知道实参的类型，同时避免了方法内不必要的运行时类型检查。

## 3.8　关闭动态类型

本书所涉及的所有元编程能力，都依赖Groovy的动态类型，但动态类型是有代价的。原本在编译时可以发现的错误被推迟到了运行时。此外，动态方法分派机制也有开销。尽管Java 7为缓解此性能问题而引入了动态调用功能，但是当Groovy在老版本的JVM上运行时，仍然存在性能影响。

可以让Groovy编译器将其类型检查从动态的不严格模式收紧到我们对静态类型编译器（如`javac`）所期待的水平。还可以权衡动态类型和元编程能力的收益，让Groovy编译器静态编译代码，以便得到更高效的字节码。

本节将介绍两个功能，一个是让Groovy在编译时执行更严格的检查，另一个是让它创建更高效的静态编译的字节码。

## 3.8.1 静态类型检查

对于编译时不存在的方法和属性，基于它们会在运行时被注入应用中的假设，我们可以使用Groovy的动态特性来调用和访问。一方面，这可以十分灵活地为应用提供高级功能，本书第三部分将予以介绍；而另一方面，愚蠢的拼写错误可能会蒙混过关，在运行时却会导致失败。当然，这些错误在我们的单元测试中很快就会浮出水面，然而如果程序中没有使用这样的动态能力，这就是不必要的负担。

其实，完全可以让Groovy自己识别正确的类型，并确保调用的方法和访问的属性在该类型上是合法的。可以使用特殊的注解@TypeChecked，让Groovy去检查这些种错误，这个注解可以用于类或单个方法上。如果用于一个类，则类型检查会在该类中所有的方法、闭包和内部类上执行。如果用于一个方法，则类型检查仅在目标方法的成员上执行。



通过例子来试用一下这个注解。首先创建一个方法，它有一个隐匿的错误，而且没有编译时保护。

**TypesAndTyping/NoCompiletimeCheck.groovy**

```
def shout(String str) {
  println "Printing in uppercase"
  println str.toUpperCase()
  println "Printing again in uppercase"
  println str.toUppercase()
}
try {
  shout('hello')
} catch(ex) {
  println "Failed..."
}
```

shout()方法接收一个String类型的形参，并调用toUpperCase()方法。在第二次调用中，有一个拼写错误。Groovy将在运行时报告一个错误。

```
Printing in uppercase
HELLO
Printing again in uppercase
Failed...
```

这段代码没有使用任何元编程，因此可以利用编译时验证的优势。下面把@TypeChecked注解加到这个方法上。

```
@groovy.transform.TypeChecked
def shout(String str) {
//...
```

一旦Groovy看到这个注解，它就会在目标代码上执行严格的检查。如果运行这个版本的代码，Groovy不会像上个版本一样走那么远；编译时就会出现错误。

```
Static type checking] - Cannot find matching method java.lang.String#toUppercase().
Please check if the declared type is right and if the method exists.
 @ line 10, column 11.
     println str.toUppercase()
             ^

1 error
```

对于使用@TypeChecked注解标记的代码，在编译时，编译器会验证方法或属性是否从属于该类。这会阻止我们使用任何元编程能力。例如，Groovy中默认可以向类中注入方法：

**TypesAndTyping/Inject.groovy**

```
def shoutString(String str) {
  println str.shout()
}

str = 'hello'
str.metaClass.shout = {-> toUpperCase() }
shoutString(str)
```

动态地向String类的实例添加了shout()方法，而且能够从shoutString()方法中调用它，在输出中可以看到：

```
HELLO
```

如果使用@TypeChecked注解shoutString()方法，编译器会阻止我们做进一步的处理。

```
@groovy.transform.TypeChecked
def shoutString(String str) {
  println str.shout() //编译时出错
}
```

虽然当静态类型检查生效时，不能直接调用动态方法，但是有一个变通方案。可以在Groovy对象上使用一个特殊的方法——invokeMethod()，10.6节将会讨论。

静态类型检查会限制使用动态方法。然而，它并没有阻止使用Groovy向JDK中的类添加的方法（参见第7章）。静态类型检查器会检查这些类中的方法和属性。它还会检查一个特殊的DefaultGroovyMethods类，其中包含了一些有用的、优雅的扩展方法。此外，它还会检查开发者能够添加的定制扩展，7.3节将予以讨论。例如，可以自由地在String上调用Groovy添加的reverse()方法。

**TypesAndTyping/Reverse.groovy**

```
@groovy.transform.TypeChecked
def printInReverse(String str) {
  println str.reverse() //没问题
}
printInReverse 'hello'
```

要利用静态类型检查，必须要指明方法和闭包的形参类型。能够在形参上调用的方法，被限制为该类型在编译时已知支持的方法。Groovy会推断闭包的返回类型，并相应地执行类型检查，所以不必担心此类细节。

与Java相比，Groovy的类型检查有一个优势。如果使用instanceOf检查类型，在使用该类型特定的方法或属性时，并不需要执行强制转换，下面的例子说明了这一点。

**TypesAndTyping/NoCast.groovy**

```
@groovy.transform.TypeChecked
def use(Object instance) {
  if(instance instanceof String)
    println instance.length() //不必强制转换
  else
    println instance
}
use('hello')
use(4)
```

我们介绍了如何让Groovy在编译时执行类型检查。如果注解的是一个完整的类，以进行静态类型检查，我们还可以使用SKIP参数去掉一些具体的方法，不对它们进行静态类型检查：

**TypesAndTyping/Optout.groovy**

```
import groovy.transform.TypeChecked
import groovy.transform.TypeCheckingMode

@TypeChecked
class Sample {
  //此处静态类型检查生效
  def method1() {
  }

  @TypeChecked(TypeCheckingMode.SKIP)
  def method2(String str) {
    str.shout()
  }
}
```

我们把method2()内的代码从编译时检查中去掉了，因此，除它之外，整个类都会执行静态类型检查。

静态类型检查旨在帮助在编译时识别出一些错误。如果没有错误，编译器为类型检查版本和无类型检查版本生成的字节码是类似的。如果想生成高效的字节码，则必须使用下一节要介绍的静态编译。

### 3.8.2　静态编译

Groovy元编程和动态类型的优点显而易见，但是这些优点需要以性能为代价。性能的下降与代码、所调用方法的个数等因素相关。当不需要元编程和动态能力时，与等价的Java代码相比，性能损失可能高达10%。Java 7的InvokeDynamic特性就旨在缓解这种痛苦，但是对于使用老版本Java的人而言，静态编译可能是个有用的特性。

我们可以关闭动态类型，阻止元编程，放弃多方法，并让Groovy生成性能足以与Java媲美的、高效的字节码。

可以使用`@CompileStatic`注解让Groovy执行静态编译。这样为目标代码生成的字节码会和`javac`生成的字节码很像。例如，不使用该注解，先来编译一下示例代码。

**TypesAndTyping/NoStaticCompile.groovy**

```
def shout1(String str) {
  println str.toUpperCase()
}
```

如果使用`groovyc`编译前面代码，然后执行`javac -p NoStaticCompile`，我们会发现，对`toUpperCase()`方法的调用是通过`CallSite()`进行的，它会处理Groovy的动态调用机制。

```
...
14:  invokeinterface  #57, 2; //InterfaceMethod
org/codehaus/groovy/runtime/callsite/CallSite.call:...
19:  invokeinterface  #61, 3; //InterfaceMethod
org/codehaus/groovy/runtime/callsite/CallSite.callCurrent:...
...
```

再使用`@CompileStatic`注解标记该方法。

**TypesAndTyping/StaticCompile.groovy**

```
@groovy.transform.CompileStatic
def shout1(String str) {
  println str.toUpperCase()
}
```

现在编译器生成了一个`invokevirtual`调用，和Java编译器所做的一样。

```
...
2: invokevirtual        #63; //Method java/lang/String.toUpperCase:()...
5: invokevirtual        #67; //Method groovy/lang/Script.println:...
...
```

如果想获得性能可以与Java媲美的代码，静态编译是很好的选择。而对于性能没那么关键，或者想使用元编程的代码，则不用考虑静态编译。

这一章介绍了类型相关的问题、优点以及Groovy的特性。当不愿意指明类型时，Groovy的动态类型如何让类型成为隐式的，以及当需要时，可以很方便地使用可选类型实现类型声明。另外，

Groovy中的方法分派大不一样，而且非常强大，还介绍了如何享用真正的多态，以及如何利用能力式设计。最后，我们还看到了在Groovy中如何选择性地关闭动态类型，以及在希望更多编译器检查或更好的性能的地方，如何获得静态类型的优势。下一章将介绍Groovy最有趣的特性之一——闭包。

第 4 章

# 使用闭包

4

当在Java中定义用于注册事件处理器的方法参数或创建短小的胶水代码时，会创建匿名内部类。该特性在Java 1.1中引入时，看上去像个不错的想法，但是没过多久，人们就意识到，它们变得非常冗长，尤其是对于那种确实非常短的单方法接口的实现而言。Groovy中的闭包就是去掉了那种冗长感的短小的匿名方法。

闭包是轻量级的，短小、简洁，而且将会是我们在Groovy中使用最多的特性之一。过去传递匿名类实例的地方，现在可以传递闭包。

闭包是从函数式编程的Lambda（λ）表达式派生而来的。根据Robert Sebesta的*Concepts of Programming Languages* [Seb04]一书，“一个Lambda表达式指定了一个函数的参数与映射”。闭包是Groovy最强大的特性之一，而且语法上非常优雅。或者如计算机科学家和函数式编程先驱Peter J. Landin所言：“（闭包是）可以帮你消化λ演算的一点语法糖。”

我们会通过Groovy JDK（GDK）大量使用闭包，因为GDK使用一些以闭包为参数的灵活且便捷的方法扩展了JDK。与其被迫创建接口和很多小型类，不如使用没那么多繁文缛节的小代码块来设计应用，这意味着代码减少了，杂乱冗余也减少了，而复用变多了。

本章将介绍闭包的创建和使用。我们将介绍如何使用闭包来优雅地实现某些设计模式。你还会了解到，闭包不是简单地替代匿名方法，它还可以变成解决有较高内存需求的问题的一种通用工具。所以这就开始了解和学习闭包吧。

## 4.1 闭包的便利性

Groovy中的闭包完全避免了代码的冗长，而且可以辅助创建轻量级、可复用的代码片段。通过对比闭包与我们所熟悉的传统解决方案在解决同样任务时的表现，就可以理解这种便利性。

### 4.1.1 传统方式

举个简单的例子：求1到某个特定的数$n$之间所有偶数的和。

下面是传统方式：

UsingClosures/UsingEvenNumbers.groovy

```
def sum(n) {
  total = 0
  for(int i = 2; i <= n; i += 2) {
    total += i
  }
  total
}
println "Sum of even numbers from 1 to 10 is ${sum(10)}"
```

sum()方法中运行了一个for循环，在偶数上迭代并求和。现在假设我们不想求和了，而是要计算从1到$n$之间所有偶数的积。代码如下：



UsingClosures/UsingEvenNumbers.groovy

```
def product(n) {
  prod = 1
  for(int i = 2; i <= n; i += 2) {
    prod *= i
  }
  prod
}
println "Product of even numbers from 1 to 10 is ${product(10)}"
```

又在偶数上进行了迭代，这次计算的是它们的积。现在，假如想得到由这些偶数值的平方所组成的集合，又该怎么办呢？返回平方值所组成数组的代码可能会写成下面这样：

UsingClosures/UsingEvenNumbers.groovy

```
def sqr(n) {
  squared = []
  for(int i = 2; i <= n; i += 2) {
    squared << i ** 2
  }
  squared
}
println "Squares of even numbers from 1 to 10 is ${sqr(10)}"
```

在前面的几个代码示例中，进行循环的代码是相同的（也是重复的）。差别在于处理的是和、积还是平方。如果想在偶数上执行一些其他操作，我们还要重复这些遍历数字的代码。我们得想办法去掉这种重复。

### 4.1.2 Groovy方式

以上三个例子产生的是不同的结果，但它们都有一个共同的任务，那就是从给定集合中挑选

出偶数。我们就从解决这一共同任务的一个函数入手。这里不是返回一个偶数的列表，而是这样：当挑选出一个偶数时，直接将其发给一个代码块来处理。现在，可以让代码块简单地打印传入的数字：

**UsingClosures/PickEven.groovy**

```
def pickEven(n, block) {
  for(int i = 2; i <= n; i += 2) {
    block(i)
  }
}

pickEven(10, { println it } )
```

pickEven()方法是一个高阶函数，即以函数为参数，或返回一个函数作为结果的函数[①]。该方法对值进行迭代（和前面一样），但不同的是它将值发送给了一个代码块。在Groovy中，我们称这种匿名代码块为闭包（Closure），Groovy设计团队使用了该术语的一个不太严格的定义[②]。

变量block保存了一个指向闭包的引用。可以像传递对象一样传递闭包。变量名没必要一定命名为block，可以使用任何合法的变量名。当调用pickEven()方法时，现在可以像前面代码中演示的那样向其发送代码块。代码块（{}内的代码）被传给形参block，就像把值10传给变量n。在Groovy中，想传递多少闭包就可以传递多少。例如，方法调用的第一个、第三个和最后一个实参都可以是闭包。如果闭包是最后一个实参，可以用下面这种优雅的语法：

**UsingClosures/PickEven.groovy**

```
pickEven(10) { println it }
```

当闭包是方法调用的最后一个实参时，可以将闭包附到方法调用上。在这种情况下，代码块看上去就像是附在方法调用上的寄生虫。不同于Java代码块，Groovy的闭包不能单独存在，只能附到一个方法上，或是赋值给一个变量。

代码块中的it是什么呢？如果只向代码块中传递一个参数,那么可以使用it这个特殊的变量名来指代该参数。如果你喜欢，也可以像下面的例子这样，用其他名字代替it：

**UsingClosures/PickEven.groovy**

```
pickEven(10) { evenNumber -> println evenNumber }
```

变量evenNumber现在指代的是在pickEven()方法内传递给该闭包的实参。

回到关于偶数的那些计算问题。我们可以使用pickEven()来求和，像这样：

---

① 参见http://c2.com/cgi/wiki?HigherOrderFunction。

② 参见http://groovy.codehaus.org/Closures+-+Formal+Definition。

UsingClosures/PickEven.groovy

```
total = 0
pickEven(10) { total += it }
println "Sum of even numbers from 1 to 10 is ${total}"
```

我们是从简单打印pickEven()生成的偶数值入手的，但是并未对函数做任何修改，就可以求和了。不像传统的方式那样重复地写一段代码，这里的代码很简洁，而且复用性更好。该函数并不限于计算这些值的和，可以复用作其他用途，比如计算积，就像下面的代码这样：

UsingClosures/PickEven.groovy

```
product = 1
pickEven(10) { product *= it }
println "Product of even numbers from 1 to 10 is ${product}"
```

除了语法上的优雅，闭包还为函数将部分实现逻辑委托出去提供了一种简单、方便的方式。



前面示例中的代码块所做的事情，要比我们更早之前看到的代码块多。它将触角伸到了pickEven()的调用者的作用域之内，使用了变量product。这是闭包的一个有趣特性。闭包是一个函数，这里变量都绑定到了一个上下文或环境中，这个函数就在其中执行。

知道了如何创建闭包，下一步将探讨如何在应用中使用闭包。

## 4.2 闭包的应用

本章会一直谈论闭包的强大与优雅，不过首先还是先探讨一下如何将闭包应用于我们的项目之中。在面对某个特定的功能或任务时，需要决定是以普通的函数或方法来实现，还是使用闭包。

闭包能够扩充、优化或增强另一段代码。比如，可以将选择对象的操作通过一个谓词或条件提炼出来，而闭包对于表达这样的谓词或条件可能很有用。另外，也可以通过闭包来使用协程（Coroutine），实现诸如迭代器或循环中的控制流转移。

闭包有两个非常擅长的具体领域：一个是辅助资源清理（参见4.5节），另一个是辅助创建内部的领域特定语言（DSL，详情参见第19章）。

普通函数在实现某个目标明确的任务时优于闭包。重构的过程是引入闭包的好时机。

一旦代码工作起来，就可以重新审视这些代码，看看闭包能否对其加以改进，使之更为优雅。要让闭包在这样的过程中浮现出来，而不是一开始就强制使用。

闭包应该保持短小，有内聚性。闭包应该设计为附到方法调用上的小段代码，只有几行。当编写使用闭包的方法时，最好不要滥用闭包的动态属性，比如在运行时确定参数的数目和类型。在调用方法时实现的闭包一定要非常简单，而且做到显而易见。

看过了使用闭包的便利性与优势，下一节来看一下闭包的不同使用方式。

## 4.3　闭包的使用方式

前面介绍了如何在定义方法调用的参数时即时创建闭包。此外，还可以将闭包赋值给变量并复用，现在就来试试看。

在下面的示例中，totalSelectValues()接受一个闭包，用于帮助确定要将哪些值用于计算中：

**UsingClosures/Strategy.groovy**

```
def totalSelectValues(n, closure) {
  total = 0
  for(i in 1..n) {
    if (closure(i)) { total += i }
  }
  total
}
print "Total of even numbers from 1 to 10 is "
println totalSelectValues(10) { it % 2 == 0 }

def isOdd = { it % 2 != 0}
print "Total of odd numbers from 1 to 10 is "
println totalSelectValues(10, isOdd)
```

totalSelectValues()方法从1迭代到$n$，它会对每个值调用闭包，以确定该值是否要用于计算中，它将选择过程委托给了该闭包。

即便在闭包中，return也是可选的。如果没有显式的return，最后一个表达式的值（可能是null）会自动返回给调用者。

在第一次调用totalSelectValues()时，将闭包内联到了方法调用中，该闭包仅选择偶数。另一方面，预先定义了要传给第二个调用的闭包。这个通过变量isOdd引用的闭包仅选择奇数。与调用时直接创建的闭包不同，这种预先定义的闭包可以在多个调用中复用。顺便插一句，不费吹灰之力，这个例子就实现了策略模式。[①]

在调用时即时创建闭包之后，我们又了解了预先定义闭包。将闭包缓存下来，以备将来之用，这种方法很有用，下面就会看到。

假设我们正在创建一个模拟器，使用它可以为设备插入不同的计算。我们想执行一些计算，但是希望使用恰当的计算程序。下面是实现该功能的一个例子：

**UsingClosures/Simulate.groovy**

```
class Equipment {
  def calculator
```

① 关于该模式的具体细节，请参考*Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]一书。

```
  Equipment(calc) { calculator = calc }
  def simulate() {
    println "Running simulation"
    calculator() // 可能还会发送参数
  }
}
def eq1 = new Equipment({ println "Calculator 1" })
def aCalculator = { println "Calculator 2" }
def eq2 = new Equipment(aCalculator)
def eq3 = new Equipment(aCalculator)

eq1.simulate()
eq2.simulate()
eq3.simulate()
```

Equipment的构造器接收闭包作为一个参数，并将其缓存到calculator属性中。simulate()方法调用该闭包来执行计算。在创建Equipment的实例eq1时，我们通过一个内联的闭包为其提供了一个计算程序（关于这种语法限制，参见2.11.5节）。如果需要复用该代码快，又该如何呢？可以将闭包保存到一个变量中，就像前面代码中的aCalculator，在创建Equipment的另外两个实例（即eq2和eq3）时就使用了它。输出说明设备使用了缓存的计算程序：

```
Running simulation
Calculator 1
Running simulation
Calculator 2
Running simulation
Calculator 2
```

Collection相关的类大量使用了闭包，要找使用闭包的例子，这是个好地方。具体细节可以参考6.2节。

介绍完如何创建和复用闭包，下一节来看一下如何向闭包传递参数。

## 4.4　向闭包传递参数

前面几节介绍了如何定义和使用闭包，这一节将探讨如何向闭包发送多个参数。

对于单参数的闭包，it是该参数的默认名称。只要知道只传入一个参数，就可以使用it。如果传入的参数多于一个，就需要通过名字一一列出了，如下面这个例子：

**UsingClosures/ClosureWithTwoParameters.groovy**

```
def tellFortune(closure) {
  closure new Date("09/20/2012"), "Your day is filled with ceremony"
}
tellFortune() { date, fortune ->
  println "Fortune for ${date} is '${fortune}'"
}
```

在调用闭包closure时，tellFortune()方法提供了两个参数：一个Date实例，一个表示运势信息的String。该闭包分别用name和fortune引用它们。符号->将闭包的参数声明与闭包主体分隔开来。运行这段代码，看一下输出[①]：

```
Fortune for Thu Sep 20 00:00:00 MST 2012 is 'Your day is filled with ceremony'
```

因为Groovy支持可选的类型，所以可以在闭包中定义参数的类型，如下面例子所示：

**UsingClosures/ClosureWithTwoParameters.groovy**

```
tellFortune() { Date date, fortune ->
  println "Fortune for ${date} is '${fortune}'"
}
```

如果为参数选择了表现力好的名字，通常可以避免定义类型。后面会看到，在元编程中，我们可以使用闭包来覆盖或替代方法，而在那种情况下，类型信息对于确保实现的正确性非常重要。

## 4.5　使用闭包进行资源清理

Java的自动垃圾收集是把双刃剑。只要我们释放了引用，就不用担心资源回收问题。但是资源何时会被自动清理并没有保证，因为这要任凭垃圾收集器处理。某些情况下，我们可能希望清理直接进行。这就是我们会在资源密集型的类中看到close()和destroy()这样的方法的原因。

不过仍然存在一个问题，使用这些类的人可能会忘记调用这些方法。闭包可以帮助确保这些方法被调用。下列代码将创建一个FileWriter，并写入一些数据，但是没有在它上面调用close()。如果运行这段代码，文件output.txt中并没有我们所输入的数据或字符。

**UsingClosures/FileClose.groovy**

```
writer = new FileWriter('output.txt')
writer.write('!')
// 忘记调用writer.close()
```

使用Groovy添加的withWriter()方法重写这段代码。当从闭包返回时，withWriter()会自动刷新（flush）并关闭这个流。

**UsingClosures/FileClose.groovy**

```
new FileWriter('output.txt').withWriter { writer ->
  writer.write('a')
} // 不再需要自己调用close()
```

现在不必担心流的关闭了；我们可以集中精力完成工作。也可以在自己的类中实现这样的便捷方法，从而使类的使用者能够开开心心且卓有成效。例如，我们有一个Resource类，希望使

① 具体输出情况和计算机的时区设置有关，所以读者运行这段代码的结果未必和书上的输出完全一致。——译者注

用者在调用它的任何实例方法之前先调用open()，使用完成时再调用close()。

这是Resource类的一个例子：

UsingClosures/ResourceCleanup.groovy

```
class Resource {
  def open() { print "opened..." }
  def close() { print "closed" }
  def read() { print "read..." }
  def write() { print "write..." }
  //...
```

下面是一个使用该类的例子：

UsingClosures/ResourceCleanup.groovy

```
def resource = new Resource()
resource.open()
resource.read()
resource.write()
```

遗憾的是，类的使用者忘记了调用close()，资源没有关闭。从下面的输出中可以看到：

```
opened...read...write...
```

这里闭包可以帮得上忙，可以使用Execute Around Method模式（见后文说明）来处理这个问题。

创建一个名为use()的静态方法：

UsingClosures/ResourceCleanup.groovy

```
def static use(closure) {
  def r = new Resource()
  try {
    r.open()
    closure(r)
  } finally {
   r.close()
  }
}
```

这个静态方法中创建了一个Resource实例，然后在该实例上调用open()，调用闭包，最后调用close()。这里还使用try-finally对调用加以保护，所以即使调用闭包时抛出异常，也会正常调用close()。

**Execute Around Method模式**

如果有一对必须连续执行的动作，比如打开和关闭，我们就可以使用Execute Around Method模式，这是一个Smalltalk模式，Kent Beck的*Smalltalk Best Practice Patterns* [Bec96]一书中曾经讨论过。编写一个Execute Around方法，它接收一个块作为参数。在这个方法中，把对该块的调用夹到对那对方法的调用之间。即先调用第一个方法，然后调用该块，最后调用第二个方法。方法的使用者不必担心这对动作，它们会自动被调用。我们甚至可以在Execute Around方法内处理异常。

来看一下类的使用者如何使用它：

**UsingClosures/ResourceCleanup.groovy**

```
Resource.use { res ->
  res.read()
  res.write()
}
```

下面是调用`use()`方法的输出，闭包会自动执行：

```
opened...read...write...closed
```

多亏闭包，现在`close()`的调用是自动的、确定性的，而且会在恰当的时机调用。我们可以将注意力集中于应用领域及其内在复杂性上，让类库去处理诸如确保文件I/O的清理等系统级任务。

学习完如何创建闭包，以及如何将其传递给函数和类，下一节将介绍函数和闭包如何交互。

## 4.6　闭包与协程

调用一个函数或方法会在程序的执行序列中创建一个新的作用域。我们会在一个入口点（方法最上面的语句）进入函数。在方法完成之后，回到调用者的作用域。

协程（Coroutine）则支持多个入口点，每个入口点都是上次挂起调用的位置。我们可以进入一个函数，执行部分代码，挂起，再回到调用者的上下文或作用域内执行一些代码。之后我们可以在挂起的地方恢复该函数的执行。正如Donald E. Knuth所言，“与主例程和子例程之间的不对称关系不同，协程之间是完全对称的，可以互相调用。[①]”

协程对于实现某些特殊的逻辑或算法非常方便，比如用在生产者－消费者问题中。生产者会接收一些输入，对输入做一些初始处理，通知消费者拿走处理过的值做进一步计算，并输出或存储结果。消费者处理它的那部分工作，完成之后通知生产者以获取更多输入。

① *The Art of Computer Programming: Fundamental Algorithms* [Knu97]

在Java中，wait()和notify()与多线程结合使用，可以协助实现协程。闭包会让人产生“协程是在一个线程中执行”的感觉（或者说错觉）。

例如，看一下这段代码：

**UsingClosures/Coroutine.groovy**

```groovy
def iterate(n, closure) {
  1.upto(n) {
    println "In iterate with value ${it}"
    closure(it)
  }
}

println "Calling iterate"
total = 0
iterate(4) {
  total += it
  println "In closure total so far is ${total}"
  }
println "Done"
```

在这段代码中，控制流在iterate()方法和闭包中来回转移：

```
Calling iterate
In iterate with value 1
In closure total so far is 1
In iterate with value 2
In closure total so far is 3
In iterate with value 3
In closure total so far is 6
In iterate with value 4
In closure total so far is 10
Done
```

每次调用闭包，我们都会从上一次调用中恢复total的值。执行序列如图4-1所示，我们在两个函数的上下文中来回切换。

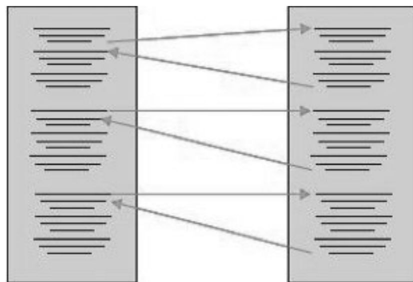


图4-1 一个协程的执行序列

介绍了函数与闭包如何交互，下一节将介绍如何改变闭包以及如何转换闭包的形参。

## 4.7　科里化闭包

闭包可能不接受任何形参，也可能接受多个形参。每次调用一个闭包时，它会期望我们为其每一个形参传入相应的实参。然而，如果在多次调用之间，有一个或多个实参是相同的，传参就会变得枯燥乏味。预先绑定一些闭包形参可以缓解这种痛苦。

带有预绑定形参的闭包叫做科里化闭包（Curried Closure）。虽然英文单词curry有“咖喱”之意，但科里化闭包与我最喜爱的印度菜并没有什么关系。（术语“科里化”源自对Lambda演算作出重要贡献的著名数学家Haskell B. Curry的名字，Christopher Strachey、Moses Schönfinkel和Friedrich Ludwig创造了这一术语。具体概念则是由Gottlob Frege发明的。）当对一个闭包调用`curry()`时，就是要求预先绑定某些形参。在预先绑定了一个形参之后，调用闭包时就不必重复为这个形参提供实参了。如图4-2所示，方法调用现在可以接受较少的参数。这有助于去掉方法调用中的冗余或重复，从下面的例子可以看出。

**UsingClosures/Currying.groovy**

```groovy
def tellFortunes(closure) {
  Date date = new Date("09/20/2012")
  //closure date, "Your day is filled with ceremony"
  //closure date, "They're features, not bugs"
  // 可以通过科里化避免重复发送date
  postFortune = closure.curry(date)
  postFortune "Your day is filled with ceremony"
  postFortune "They're features, not bugs"
}
tellFortunes() { date, fortune ->
  println "Fortune for ${date} is '${fortune}'"
}
```

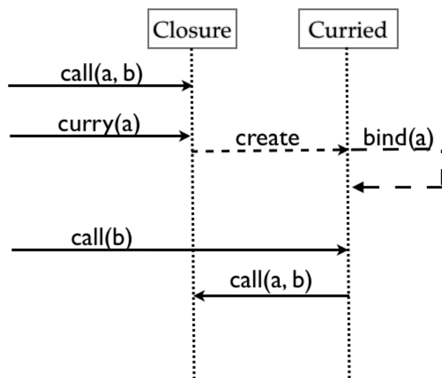



图4-2　对一个闭包进行科里化

`tellFortunes()`方法多次调用了一个闭包。该闭包接受两个形参。因此，每次调用

tellFortunes()时都要提供第一个参数date。作为一种选择，可以以date作为一个参数来调用curry()方法，实现形参date的科里化。postFortune保存着科里化之后的闭包的引用，它已经预先绑定了date的值。

现在可以调用科里化闭包了，只需要传入原来闭包的第二个形参（fortune）。科里化闭包负责把fortune和预先绑定的形参date发送给原来的闭包：

```
Fortune for Thu Sep 20 00:00:00 MST 2012 is 'Your day is filled with ceremony'
Fortune for Thu Sep 20 00:00:00 MST 2012 is 'They're features, not bugs'
```

可以使用curry()方法科里化任意多个形参，但这些形参必须是从前面开始的连续若干个。也就是说，如果有n个形参，我们可以任意科里化前k个，其中0 <= k <= n。

如果想科里化后面的形参，可以使用rcurry()方法。如果想科里化位于形参列表中间的形参，可以使用ncurry()[①]方法，传入要进行科里化的形参的位置，同时提供相应的值。

科里化是一种变换，将一个接受多个形参的函数变成了一个接受较少（通常是一个）形参的函数。函数f(X,Y) -> Z上的科里化函数被定义为curry(f): X -> (Y -> Z)。科里化有助于简化数学证明方法。就我们的目的而言，在Groovy中，科里化可以减少代码中的噪音。

在形参这个主题之后，我们将学习如何确定一个闭包是否存在，以及如何确定一个闭包可能接收的形参的数目和类型。

## 4.8 动态闭包

可以确定一个闭包是否已经提供。如果尚未提供，比如说是一个算法，我们可以决定使用该算法的一个默认实现来代替调用者未能提供的特殊实现。下面是确定一个闭包是否存在的例子：

**UsingClosures/MissingClosure.groovy**

```groovy
def doSomeThing(closure) {
  if (closure) {
    closure()
  } else {
    println "Using default implementation"
  }
}

doSomeThing() { println "Use specialized implementation" }

doSomeThing()
```

① ncurry()有两个版本：public Closure<V> ncurry(int n,Object argument)和public Closure<V> ncurry(int n,Object... arguments)，前者可以科里化位于位置n的形参，后者则可以科里化从位置n开始的多个形参。具体用法可以参考文档：http://groovy.codehaus.org/api/groovy/lang/Closure.html#ncurry%28int,%20java.lang.Object...%29。
——译者注

这段代码会确定一个闭包是不是提供了，同时进行相应处理：

```
Use specialized implementation
Using default implementation
```

在传递参数时也有很多灵活性。可以动态地确定一个闭包期望的参数的数目和类型。假设我们使用一个闭包来计算销售税额。税额取决于销售额和税率。再假设该闭包可能需要我们提供税率，也可能不需要。下面是检查参数数目的一个例子：

**UsingClosures/QueryingClosures.groovy**

```
def completeOrder(amount, taxComputer) {
  tax = 0
  if (taxComputer.maximumNumberOfParameters == 2) {// 期望传入税率
    tax = taxComputer(amount, 6.05)
  } else {// 使用默认税率
    tax = taxComputer(amount)
  }
  println "Sales tax is ${tax}"
}
completeOrder(100) { it * 0.0825 }
completeOrder(100) { amount, rate -> amount * (rate/100) }
```

maximumNumberOfParameters属性（或getMaximumNumberOfParameters()方法）告诉我们给定的闭包接受的参数个数。利用这个方法，我们在computeOrder()方法中确定了给定闭包接受的参数个数，并以此确定是否需要提供税率。这可以帮助我们使用正确的参数数目调用给定闭包，在输出中可以看到：

```
Sales tax is 8.2500
Sales tax is 6.0500
```

除了参数个数，还可以使用parameterTypes属性（或getParameterTypes()方法）获知这些参数的类型。下面这个例子用于检查所提供闭包的参数的类型：

**UsingClosures/ClosuresParameterTypes.groovy**

```
def examine(closure) {
  println "$closure.maximumNumberOfParameters parameter(s) given:"
  for(aParameter in closure.parameterTypes) { println aParameter.name }

  println "--"
}

examine() { }
examine() { it }
examine() {-> }
examine() { val1 -> }
examine() {Date val1 -> }
```

```
examine() {Date val1, val2 -> }
examine() {Date val1, String val2 -> }
```

运行这段代码，看一下参数的个数以及所报告的类型：

```
1 parameter(s) given:
java.lang.Object
--
1 parameter(s) given:
java.lang.Object
--
0 parameter(s) given:
--
1 parameter(s) given:
java.lang.Object
--
1 parameter(s) given:
java.util.Date
--
2 parameter(s) given:
java.util.Date
java.lang.Object
--
2 parameter(s) given:
java.util.Date
java.lang.String
--
```

即便一个闭包没有使用任何形参，就像`{}`或`{ it }`中这样，其实它也会接受一个参数（名字默认为`it`）。如果调用者没有向闭包提供任何值，则第一个形参（`it`）为`null`。如果希望闭包完全不接受任何参数，必须使用`{-> }`这种语法：在`->`之前没有形参，说明该闭包不接受任何参数。

利用`maximumNumberOfParameters`和`parameterTypes`属性，我们可以动态地检查闭包，而且在实现某些逻辑时有了更大的灵活性。

谈到检查对象，闭包内的`this`有什么意义呢？下面我们来看一下。

## 4.9　闭包委托

Groovy中的闭包远远超越了简单的匿名方法，本章余下的几节将关注它们还有哪些强大的功能。Groovy的闭包支持方法委托，而且提供了方法分派能力，这点和JavaScript对原型继承（prototypal inheritance）的支持很像。我们来了解一下其背后的魔法，以及如何好好利用这一点。

`this`、`owner`和`delegate`是闭包的三个属性，用于确定由哪个对象处理该闭包内的方法调用。一般而言，`delegate`会设置为`owner`，但是对其加以修改，可以挖掘出Groovy的一些非常好的元编程能力。我们来观察一下闭包的这三个属性：

**UsingClosures/ThisOwnerDelegate.groovy**

```
def examiningClosure(closure) {
  closure()
}

examiningClosure() {
  println "In First Closure:"
  println "class is " + getClass().name
  println "this is " + this + ", super:" + this.getClass().superclass.name
  println "owner is " + owner + ", super:" + owner.getClass().superclass.name
  println "delegate is " + delegate +
            ", super:" + delegate.getClass().superclass.name

  examiningClosure() {
    println "In Closure within the First Closure:"
    println "class is " + getClass().name
    println "this is " + this + ", super:" + this.getClass().superclass.name
    println "owner is " + owner + ", super:" + owner.getClass().superclass.name
    println "delegate is " + delegate +
              ", super:" + delegate.getClass().superclass.name
  }
}
```

在第一个闭包内，取到了该闭包的一些具体信息，确定了this、owner和delegate分别指向的是什么。之后又在第一个闭包内调用了examiningClosure()方法，同时向它传递了另一个闭包。因为第二个闭包是在第一个闭包内定义的，所以第一个闭包成了第二个闭包的owner。在第二个闭包内，我们再次打印了这些具体信息。这段代码的输出如下：

```
In First Closure:
class is ThisOwnerDelegate$_run_closure1
this is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
owner is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
delegate is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
In Closure within the First Closure:
class is ThisOwnerDelegate$_run_closure1_closure2
this is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
owner is ThisOwnerDelegate$_run_closure1@15c330aa, super:groovy.lang.Closure
delegate is ThisOwnerDelegate$_run_closure1@15c330aa, super:groovy.lang.Closure
```

前面的代码示例和相应输出说明，闭包被创建成了内部类。此外还说明，delegate被设置为owner。某些Groovy函数会修改delegate，以执行动态路由，比如with()函数。闭包内的this指向的是该闭包所绑定的对象（正在执行的上下文）。在闭包内引用的变量和方法都会绑定到this，它负责处理任何方法调用，以及对任何属性或变量的访问。如果this无法处理，则转向owner，最后是delegate。下图说明了这种解析顺序。

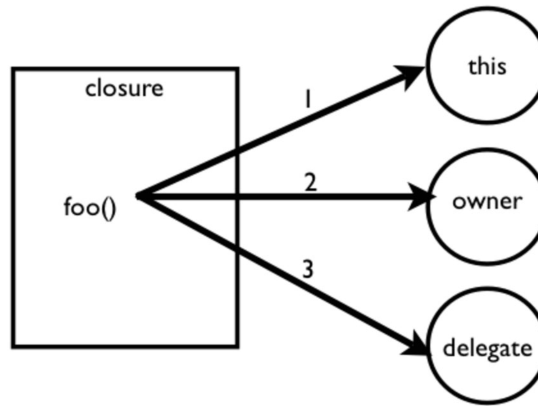



图4-3 闭包内的方法调用解析顺序



下面通过一个例子进一步观察方法解析：

**UsingClosures/MethodRouting.groovy**

```
class Handler {
  def f1() { println "f1 of Handler called ..."}
  def f2() { println "f2 of Handler called ..."}
}

class Example {
  def f1() { println "f1 of Example called ..."}
  def f2() { println "f2 of Example called ..."}

  def foo(closure) {
    closure.delegate = new Handler()
    closure()
  }
}

def f1() { println "f1 of Script called..." }

new Example().foo {
  f1()
  f2()
}
```

在这段代码中，闭包内的方法调用首先被路由到闭包的上下文对象——`this`。如果没找到这些方法，则路由到`delegate`：

```
f1 of Script called...
f2 of Handler called ...
```

前面的示例中设置了闭包的`delegate`属性。不过这会有副作用，尤其是当该闭包还被用于

其他的函数或线程时。如果完全肯定该闭包不会用在别的地方，那自然可以设置delegate；如果它用在了其他地方，为避免副作用，还请复制该闭包，在副本上设置delegate，并使用副本。Groovy为实现这一点提供了一个便捷方法，不再需要执行下面几行语句：

```
def clone = closure.clone()
clone.delegate = handler
clone()
```

借助一个特殊的with()方法，可以一次性执行这三个步骤：

```
handler.with closure
```

19.7节介绍了如何将本节介绍的概念应用于构建DSL。此外，还可以参考7.1节和13.2节。ExpandoMetaClass使用delegate来代理类的方法。

Groovy的闭包可不仅仅是简单的胶水代码。除了动态方法分派能力，它们还有一些有趣的便捷方法，下一节我们将看到。

## 4.10　使用尾递归编写程序

使用递归会遇到一些较为常见的问题，而借助Groovy中的闭包，我们可以在获得递归之优势的同时避免这些问题。

递归可以通过子问题的解决方案来解决主干问题。递归解决方案的魅力在于非常简洁，而且只需利用输入规模较小的相同问题的解决方案来组合出最终解决方案，这点很酷。尽管存在这些优势，但是程序员往往对递归解决方案敬而远之。在输入规模较大的情况下，潜在的StackOverflowError威胁，使得最优秀的程序员都有可能望而却步。

下面是使用一个简单递归实现的极为简化的阶乘函数，我们再熟悉不过了。

**UsingClosures/simpleFactorial.groovy**

```
def factorial(BigInteger number) {
  if (number == 1) 1 else number * factorial(number - 1)
}

try {
  println "factorial of 5 is ${factorial(5)}"
  println "Number of bits in the result is ${factorial(5000).bitCount()}"
} catch(Throwable ex) {
  println "Caught ${ex.class.name}"
}
```

5的阶乘是个比较小的值，但5000的阶乘就非常大了。如果计算成功，应该看到的结果是120以及5000的阶乘这个很大的数所包含的位数。当运行程序时，我们发现JVM被数量这么大的递归调用卡住了：

```
factorial of 5 is 120
Caught java.lang.StackOverflowError
```

如果将该函数写为迭代形式，则不会遇到这样的资源限制。但是递归是这么酷，而且富有表现力，要是对资源使用再友好一些，那该多好啊……

*Structure and Interpretation of Computer Programs* [AS96]这本书很不错，书中几位作者论述了一种方法，可以优雅地解决此问题。他们提出，借助编译器优化技术和语言支持，递归程序可以转换为迭代过程。使用这种变换，可以编写出性能极高而且非常优雅的代码，同时还能获得简单迭代的效率优势。Groovy语言通过闭包上的一个特殊的`trampoline()`方法提供了这种技巧。

要使用该特性，首先必须将递归函数实现为闭包：

**UsingClosures/trampolineFactorial.groovy**

```
def factorial

factorial = { int number, BigInteger theFactorial ->
  number == 1 ? theFactorial :
    factorial.trampoline(number - 1, number * theFactorial)
}.trampoline()

println "factorial of 5 is ${factorial(5, 1)}"
println "Number of bits in the result is ${factorial(5000, 1).bitCount()}"
```

这里定义了一个名为`factorial`的变量，并将一个闭包赋给它。该闭包接受两个参数：一个是`number`，要计算的就是它的阶乘；一个是`theFactorial`，它表示通过这个递归计算出的部分结果。在闭包中，如果给定的`number`是1，就返回`theFactorial`的值作为结果。 否则，就通过调用`trampoline()`方法递归地调用该闭包。将`number - 1`作为第一个参数传给该方法，以缩减计算范围。第二个参数是到目前为止计算出的部分阶乘结果。

`factorial`变量本身被赋的就是在闭包上调用`trampoline()`方法的结果。

Groovy中的*尾递归*实现非常出彩，没有对语言本身做任何修改就实现了。当我们调用`trampoline()`方法时，该闭包会直接返回一个特殊类`TrampolineClosure`的一个实例。当我们向该实例传递参数时，比如像`factorial(5, 1)`中这样，其实是调用了该实例的`call()`方法。该方法使用了一个简单的`for`循环来调用闭包上的`call`方法，直到不再产生`TrampolineClosure`的实例。这种简单的技术在背后将递归调用转换成了一个简单的迭代。

这种递归之所以叫作尾递归，是因为方法中最后的表达式或者是结束递归，或者是调用自身。相反，在直接递归计算阶乘时，最后的表达式调用的是*，即乘法操作符。

运行新的尾递归版本的`factorial()`，我们会发现，这个版本已经没有直接递归版本的缺点：

```
factorial of 5 is 120
Number of bits in the result is 24654
```

4

trampoline()方法帮我们在享受递归之强大的同时避免了其缺点。这是很大的进步，但是在这一过程中，简洁性也失去了。现在不再是像factorial(5)这样简单地调用该方法，我们必须提供一个额外的参数，比如factorial(5, 1)。除了这种额外的负担，还很容易出错，比如我们可能为第二个参数提供了别的值，而不是1。

这个问题有个权宜之计——可以为闭包的第二个参数定义一个默认值，就像这样：BigInteger theFactorial = 1。调用者现在可以省略第二个参数，但还是无法防止他们提供错误的值。可以将闭包封装到一个函数内来解决这个问题。

**UsingClosures/trampoline.groovy**

```
def factorial(int factorialFor) {
  def tailFactorial
  tailFactorial = { int number, BigInteger theFactorial = 1 ->
    number == 1 ? theFactorial :
      tailFactorial.trampoline(number - 1, number * theFactorial)
  }.trampoline()
  tailFactorial(factorialFor)
}
println "factorial of 5 is ${factorial(5)}"
println "Number of bits in the result is ${factorial(5000).bitCount()}"
```

这里定义了一个函数factorial()，同时在该函数内定义了尾递归闭包，并调用了trampoline()方法。在该函数的最后，我们调用了闭包，并将想要计算阶乘的数传给它。闭包的第二个参数会被传入默认值1。

某些语言是通过编译技术来实现尾递归优化的。与它们不同，Groovy是通过闭包上的一个便捷方法实现的。这避免了对语言及其语法的整体影响，在减少内存占用的同时使编程变得更为优雅。不过使用trampoline()特性还是有个问题——它的性能要比简单递归或纯粹的迭代差一些。对于较大的输入值，这可能是个很大的限制。（不需要任何编译器技巧的实现固然是好事，但是希望不影响性能却并不现实。）要使trampoline()适用于较大的输入值，或者需要大大改进该实现的性能，或者需要一种编译器变换来改进效率。

下一节将介绍闭包对一种特殊类型的递归算法的运行时影响。

## 4.11　使用记忆化改进性能

我曾经看过一场鸟类表演，表演者让一个小伙子和一只鹦鹉进行算数比赛。这只鹦鹉可以快速地回答出“200乘以50是多少”之类问题，观众很开心，不过这个小伙子很懊恼。在鹦鹉又答对了接下来的一些问题之后，表演者向大家透露了其中的秘密——对于这些照本宣科的问题，鹦鹉只是在简单地重复所记住的答案。本节会使用一种类似的技术，不过我们不会称其为“记住”，而是继续本领域会为概念起个奇怪名字的传统，使用“记忆化”（Memoization）这个术语。

上一节介绍了一种可以使递归调用更高效地使用内存的技巧。递归本质上是一种使用子问题的解决方案来解决问题本身的方式。这种技巧有一个变种（被奇怪地命名为动态规划）[①]，将问题分解为可以多次重复解决的若干部分。在执行期间，我们将子问题的结果保存下来，当调用到重复的计算时，只需要简单地使用保存下来的结果。这就避免了重复运行，因此极大地减少了计算时间。记忆化可以将一些算法的计算时间复杂度从输入规模（`n`）的指数级（`O(k^n)`）降低到只有线性级（`O(n)`）。

为帮助理解记忆化这一概念以及Groovy中的相关设施，我们来看一下卖杆这种业务。不同长度的杆零售价格不同。我们会批发某一特定长度的杆，比如说27英寸（1英寸=2.54厘米），然后将其分割成长度不一的杆销售，以实现收入最大化。

首先使用简单递归来解决这个问题，然后再使用记忆化。因为记忆化是要减少计算时间，所以我们需要一种手段来度量所消耗的时间。就从一个可以帮助我们度量时间的小函数入手吧。

**UsingClosures/rodCutting.groovy**

```
def timeIt(length, closure) {
  long start = System.nanoTime()
  println "Max revenue for $length is ${closure(length)}"
  long end = System.nanoTime()
  println "Time taken ${(end - start)/1.0e9} seconds"
}
```

`timeIt()`方法会报告给定闭包运行所消耗的时间，并报告对于给定长度的杆，我们可以期望的最大收入，这里的最大收入是该闭包输出的。

作为示例，下面使用一个数组为各种长度的杆[从0英寸到30英寸（不包含）]定义一组零售价格。之所以要包含长度为0的杆，是为了弥合基于0的数组索引所带来的问题：[②]

**UsingClosures/rodCutting.groovy**

```
def rodPrices = [0, 1, 3, 4, 5, 8, 9, 11, 12, 14,
  15, 15, 16, 18, 19, 15, 20, 21, 22, 24,
  25, 24, 26, 28, 29, 35, 37, 38, 39, 40]

def desiredLength = 27
```

`rodPrices`变量指向一个列表，其中保存了不同长度的杆的零售价格。变量`desiredLength`保存的是我们想要将其分割销售以实现收入最大化的原始杆的长度。

根据给定的零售价格，如果直接销售27英寸长的杆，收入为38美元。如果将其分割成分别长

① 动态规划的英文表达是Dynamic Programming，如果不清楚其背景，可能会与本书所谈的动态特性编程相混淆，所以说命名有点奇怪。——译者注

② 加入这个0之后，我们就可以以杆长度作为索引值来获得其零售价格了，比如`rodPrices[1]`表示的就是长度为1的杆的价格，也就是1，依此类推。——译者注

1英寸和26英寸的杆来销售，收入还是38美元，因此实际上不值得分割。如果分割成长4英寸和23英寸的两段，还会损失一些钱。如果分割成六段，五段5英寸长的，一段2英寸长的，总共会带来43美元的销售收入，超过了前面的38美元。人工计算需要相当长的时间。如果给定了一个任意的长度和价格，我们希望有一个程序能快速地计算出使得收入最大化的最优分割方案。

对于一个给定的长度，下面将要编写的代码会给出最大收入和实现最大收入的分割方法，也就是分割后各杆的长度。

先从一个保存价格和分割后各段长度的类入手：RevenueDetails。

**UsingClosures/rodCutting.groovy**

```
@groovy.transform.Immutable
class RevenueDetails {
  int revenue
  ArrayList splits
}
```

要找出一个最优的最大收入，需要尝试各种组合。对于一个给定的长度，其最大收入，就是以不同方式对杆进行分割，分别计算所得收入，然后求出的最大值。例如，我们可以先将杆分割为一段2英寸的和一段25英寸的。不过这种分割方式可以获得的最大收入，不是这两种长度的零售价格之和，而是它们分别可以获得的最大收入之和。可见解决方案中已经浮现出递归结构。

为确定25英寸的最大收入，我们将其分割成更小的段，然后依次重复地计算较小的段（比如长2英寸的段）的最大收入。为节省时间，可以把这种重复计算记录下来，我们一会就会看到。

首先为杆切割问题实现简单递归方案。

**UsingClosures/rodCutting.groovy**

```
def cutRod(prices, length) {
  if(length == 0)
    new RevenueDetails(0, [])
  else {
    def maxRevenueDetails = new RevenueDetails(Integer.MIN_VALUE, [])
    for(rodSize in 1..length) {
      def revenueFromSecondHalf = cutRod(prices, length - rodSize)
      def potentialRevenue = new RevenueDetails(
        prices[rodSize] + revenueFromSecondHalf.revenue,
        revenueFromSecondHalf.splits + rodSize)
      if(potentialRevenue.revenue > maxRevenueDetails.revenue)
        maxRevenueDetails = potentialRevenue
    }
    maxRevenueDetails
  }
}

timeIt desiredLength, { length -> cutRod(rodPrices, length) }
```

cutRod()方法接受prices和length这两个参数，返回一个RevenueDetails实例，其中保存了给定长度的最大收入和各段的可能长度。如果length为0，则递归结束。给定一个长度，我们会尝试尽可能多的分割组合：1和length - 1、2和length - 2以及3和length - 3，诸如此类，同时求出每种组合的最大收入。对于每对长度，递归地调用cutRod()方法。

这是一种简单的递归，每次调用该方法都会执行全部计算。为同样的长度重复调用该方法，会反复地重新计算结果。以27英寸长的杆为例，运行这段代码，看一下最大收入、最优分割长度以及这段代码求得这些结果所消耗的时间。

```
Max revenue for 27 is RevenueDetails(43, [5, 5, 5, 5, 5, 2])
Time taken 162.89431500 seconds
```

要获得43美元的最大收入，需要将杆分割成六段。这个程序用了两分多钟才求出该结果。我们可以使用记忆化改进其速度。出人意料的是，只需要很小的改动就能实现——将该函数转换为一个闭包，然后在闭包上调用memoize()方法：



**UsingClosures/rodCutting.groovy**

```groovy
def cutRod
cutRod = { prices, length ->
  if(length == 0)
    new RevenueDetails(0, [])
  else {
    def maxRevenueDetails = new RevenueDetails(Integer.MIN_VALUE, [])
    for(rodSize in 1..length) {
      def revenueFromSecondHalf = cutRod(prices, length - rodSize)
      def potentialRevenue = new RevenueDetails(
        prices[rodSize] + revenueFromSecondHalf.revenue,
        revenueFromSecondHalf.splits + rodSize)
      if(potentialRevenue.revenue > maxRevenueDetails.revenue)
        maxRevenueDetails = potentialRevenue
    }
    maxRevenueDetails
  }
}.memoize()

timeIt desiredLength, { length -> cutRod(rodPrices, length) }
```

在将函数转换为闭包，并在其上调用了memoize()方法之后，我们将结果保存到了cutRod变量中。通过这些步骤，我们创建了Memoize类的一个专用实例。该实例中有一个指向所提供闭包的引用，还有一个结果的缓存。当我们调用该闭包时，该实例会在返回结果之前将响应缓存下来。在随后的调用中，如果对应某个参数已经有了相应的缓存下来的结果值，则返回该值。

运行修改版的代码，并确认所得的最大收入和各段长度与之前版本相同。该版本应该会节省很多时间：

```
Max revenue for 27 is RevenueDetails(43, [5, 5, 5, 5, 5, 2])
Time taken 0.01171600 seconds
```

记忆化版本产生的结果相同，但是在消耗的时间上，前面的简单递归版本花了2分多钟，而该版本只用了1/100秒。

记忆化技术是以空间换取速度。我们看到了速度的提升。这种技术消耗的空间取决于使用不重复的参数值调用递归方法的次数。对于较大的问题规模，内存需求可能会急剧增长。Groovy对此非常敏感，所以为我们提供了一些选项。简单地调用`memoize()`，该方法会使用一个没有限制的缓存。我们可以使用`memoizeAtMost()`方法代替它。该方法会限制缓存的大小，而且当达到该限制时，最近最少使用（Least Recently Used，LRU）的值会从缓存中移出，以容纳新的值。

还可以使用诸如`memoizeAtLeast()`和`memoizeAtLeastBetween()`之类的变种，前者可以设置缓存的下限，而后者可以同时设置下限和上限。除了管理缓存，`memoize()`函数的实现还提供了线程安全性；我们可以安全地从多个线程访问缓存。

我们看到，动态类型的Groovy使得动态规划实现起来非常轻松。

本章介绍了Groovy中最重要的概念之一——闭包，这也是将来会反复用到的概念。我们现在已经知道如何在动态上下文中使用闭包，也理解了闭包如何分派方法调用。后面的章节会介绍一些出现闭包的例子，因此我们有的是机会欣赏它的魅力。

在编写程序时，字符串司空见惯，而Groovy为处理它们提供了极大的便利性和灵活性。在下一章中，我们将探讨Groovy提供的涵盖从创建字符串到格式化表达式等不同功能的字符串处理设施。

第 5 章

# 使用字符串

我们都知道，在Java中使用字符串是种痛苦。本以为像字符串这么基础的东西应该更容易些，但事与愿违。基本的字符串操作，以字符串形式表示多个变量或表达式的求值，甚至连创建跨多行的字符串这么简单的事情，都很费劲。Groovy来拯救我们了！它终结了这些字符串操作的痛苦。通过提供特殊的操作符，利用正则表达式对字符串进行模式匹配也变得更容易了。

本章就依次介绍一下Groovy字符串的基础知识。



## 5.1 字面常量与表达式

Groovy中可以使用单引号创建字符串字面常量，比如`'hello'`。而在Java中，`'a'`是一个`char`，`"a"`才是一个`String`对象。Groovy中没有这样的分别。在Groovy中，二者都是`String`类的实例。如果想显式地创建一个字符，只需要输入`'a' as char`。当然，如果有任何方法调用需要的话，Groovy可能会隐式地创建`Character`对象。

对于字符串字面常量中可以放什么，Groovy也很灵活。只要你想，双引号都可以放到字符串中：

**WorkingWithStrings/Literals.groovy**

```
println 'He said, "That is Groovy"'
```

由输出可见，Groovy处理得相当好：

```
He said, "That is Groovy"
```

来看看使用单引号创建的对象的类型：

**WorkingWithStrings/Literals.groovy**

```
str = 'A string'
println str.getClass().name
```

从输出中可以看到，该对象就是常见的`String`：

```
java.lang.String
```

Groovy会把使用单引号创建的String看作一个纯粹的字面常量。因此，如果在里面放了任何表达式，Groovy并不会计算它们；相反，它就是按所提供的字面内容来使用它们。要对String中的表达式进行求值运算，则必须使用双引号，一会儿就会介绍。

**WorkingWithStrings/Literals.groovy**

```
value = 25
println 'The value is ${value}'
```

从输出中可以看到，Groovy没有对value进行求值计算：

```
The value is ${value}
```

Java的String是不可变的，Groovy也信守这种不可变性。一旦创建了一个String实例，就不能通过调用更改器等方法修改其内容了。可以使用[]操作符读取一个字符；不过不能修改，从下面的代码中可以看到：

**WorkingWithStrings/Literals.groovy**

```
str = 'hello'
println str[2]
try {
  str[2] = '!'
} catch(Exception ex) {
  println ex
}
```

尝试修改String导致了一个错误：

```
l
groovy.lang.MissingMethodException: No signature of method:
  java.lang.String.putAt() is applicable for argument types:
  (java.lang.Integer, java.lang.String) values: [2, !]
...
```

可以使用双引号（""）或正斜杠（//）创建一个表达式[①]。不过，双引号经常用于定义字符串表达式，而正斜杠则用于正则表达式。下面是一个创建表达式的例子：

**WorkingWithStrings/Expressions.groovy**

```
value = 12
println "He paid \$${value} for that."
```

Groovy会计算该表达式，我们在输出中会看到：

```
He paid $12 for that.
```

变量value在字符串内被计算求值了。这里使用了转义字符（\）来打印$符号，因为Groovy

① 这里指的是一前一后两个正斜杠，将字符串内容包围在内。——译者注

会将$符号用于嵌入表达式。如果定义字符串时使用的是正斜杠，而非双引号，则不必转义$。如果表达式是一个像value这样的简单变量名，或者是一个简单的属性存取器（accessor），则包围表达式的{}是可选的。因此，我们可以把语句println "He paid \$${value} for that."写作println "He paid \$$value for that."或println (/He paid $$value for that/)。尝试去掉表达式中的{}，看看Groovy是否会报错。需要的时候我们总是可以加上的。

Groovy支持惰性求值（lazy evaluation），即把一个表达式保存在一个字符串中，稍后再打印。来看一个例子：

**WorkingWithStrings/Expressions.groovy**

```
what = new StringBuilder('fence')
text = "The cow jumped over the $what"
println text

what.replace(0, 5, "moon")
println text
```

通过输出看看Groovy是如何解析该表达式的：

```
The cow jumped over the fence
The cow jumped over the moon
```



当打印text中的字符串表达式时，使用的是what所指对象的当前值。因此，第一次打印text时，得到的是“The cow jumped over the fence”。在修改了StringBuilder中的值之后，再打印该字符串表达式时，我们并没有修改text的内容，但得到的输出不同，这次是来自流行的儿歌*Hey Diddle Diddle*中的那句“The cow jumped over the moon”。

从这种行为可以看出，使用单引号创建的字符串和使用双引号或正斜杠创建的字符串不同。前者是普通的java.lang.String，而后者有些特殊。Groovy的开发者以他们奇怪的幽默感称其为GString，即Groovy字符串的简称。下面看一下使用不同语法创建的对象的类型：

**WorkingWithStrings/Expressions.groovy**

```
def printClassInfo(obj) {
  println "class: ${obj.getClass().name}"
  println "superclass: ${obj.getClass().superclass.name}"
}

val = 125
printClassInfo ("The Stock closed at ${val}")
printClassInfo (/The Stock closed at ${val}/)
printClassInfo ("This is a simple String")
```

从输出中可以看到所创建对象的类型：

```
class: org.codehaus.groovy.runtime.GStringImpl
superclass: groovy.lang.GString
```

```
class: org.codehaus.groovy.runtime.GStringImpl
superclass: groovy.lang.GString
class: java.lang.String
superclass: java.lang.Object
```

Groovy并不会简单地因为使用了双引号或正斜杠就创建一个GString实例。它会智能地分析字符串，以确定该字符串是否可以使用一个简单的普通String蒙混过关。在这个例子中，最后一次调用printClassInfo()时，即使我们使用了双引号来创建字符串，但该参数还是一个String实例。

很快你就会熟悉Groovy中不同字符串类型的无缝互动。下一节我们将会看到，使用这些字符串时也必须慎重。

## 5.2 GString 的惰性求值问题

GString表达式的结果取决于表达式中是否使用了值或引用。如果表达式组织得不够仔细，结果可能会令人惊讶。我在学习Groovy字符串操作时就遇到过多次，现在学习这部分内容，可以避免你像我那样栽跟头。下面是上一节中工作得很好的那个例子：

**WorkingWithStrings/LazyEval.groovy**

```
what = new StringBuilder('fence')
text = "The cow jumped over the $what"
println text

what.replace(0, 5, "moon")
println text
```

这段代码的输出看上去相当合理：

```
The cow jumped over the fence
The cow jumped over the moon
```

text这个GString实例中包含了变量what。该表达式会在每次被打印时，也就是在其上调用toString()方法时求值。如果修改了what所指向的StringBuilder对象的值，在打印时会有所体现。这看上去很合理吧？

遗憾的是，如果修改的是引用what，而不是被引用对象的属性，结果将出乎意料。但如果对象是不可变的，修改引用就是很自然的做法。下面的例子说明了这个问题：

**WorkingWithStrings/LazyEval.groovy**

```
price = 684.71
company = 'Google'
quote = "Today $company stock closed at $price"
println quote
```

```
stocks = [Apple : 663.01, Microsoft : 30.95]

stocks.each { key, value ->
  company = key
  price = value
  println quote
}
```

这里使用嵌入的变量company和price在变量quote中保存了一个表达式。第一次打印时，这段代码正确打印了谷歌及其股价。我们还想使用这个表达式来打印其他一些公司的股价，为实现该功能，对stocks这个Map进行迭代。在闭包内，将公司名作为键，将股价作为值。然而，当打印quote时，如下所示，结果并不是我们想要的。在同事们挑起另一场“谷歌已经接管世界”的争论之前，必须修复这个问题。

```
Today Google stock closed at 684.71
Today Google stock closed at 684.71
Today Google stock closed at 684.71
```

先弄清楚它为什么没有按预期方式工作，才能找出解决方案。这里在定义quote这个GString时，使用了company和price两个变量，前者绑定的是值为“Google”的一个String，后者绑定的是一个Integer，其中保存高得惊人的股价。可以将company和price引用（它们指向的均为不可变对象）修改为任何想指向的其他对象，但是不能修改GString实例所绑定的内容。

5

“The cow jumping over...”可以工作，是因为修改的是GString所绑定的对象。然而这个例子中却不可行。因为不可变，所以无法修改。那如何解决呢？——让GString重新计算引用，毕竟，正如计算机科学家David Wheeler所说：“计算机科学领域的任何问题都可以通过增加一个间接层来解决。”

在修复该问题之前，先花点时间理解一下GString表达式是如何求值的。当对一个GString实例求值时，如果其中包含一个变量，该变量的值会被简单地打印到一个Writer，通常是一个StringWriter。然而，如果GString中包含的是一个闭包，而非变量，该闭包就会被调用。如果闭包接收一个参数，GString会把Writer对象当作一个参数发送给它。如果闭包不接收任何参数，GString会简单地调用该闭包，并打印我们想返回给Writer的结果。如果闭包接收的参数不止一个，调用则会失败，并抛出一个异常，所以别这么做。

下面使用这些知识来解决我们的表达式求值问题。第一次尝试：

**WorkingWithStrings/LazyEval.groovy**

```
companyClosure = { it.write(company) }
priceClosure = { it.write("$price") }
quote = "Today ${companyClosure} stock closed at ${priceClosure}"
stocks.each { key, value ->
  company = key
  price = value
```

```
  println quote
}
```

运行代码看一下输出：

```
Today Apple stock closed at 663.01
Today Microsoft stock closed at 30.95
```

输出合乎预期，但这段代码看上去还不够出色。即使最终版本不想采用这种方式，但是通过观察这个例子，还是会有两方面的收获。首先，可以看到实际发生了什么：当表达式需要求值/打印时，GString会调用闭包；其次，如果想做些计算，而不是仅仅显示一下属性的值，也知道了该怎么做。

如前文所述，如果闭包没有任何参数，可以去掉it参数，GString会使用我们返回的内容。我们已经知道如何创建一个没有参数的闭包——使用{->语法来定义。现在重构前面的代码：

**WorkingWithStrings/LazyEval.groovy**

```
companyClosure = {-> company }
priceClosure = {-> price }
quote = "Today ${companyClosure} stock closed at ${priceClosure}"
stocks.each { key, value ->
  company = key
  price = value
  println quote
}
```

这是重构版本的输出：

```
Today Apple stock closed at 663.01
Today Microsoft stock closed at 30.95
```

这个版本略胜一筹，但是我们不想单独定义闭包。对于这种简单情形，我们希望代码是自包含的；而如果有更多计算，我们也愿意编写一个单独的闭包。下面是解决该问题的自包含代码（我们称其为“苹果和微软试图接管世界”问题）：

**WorkingWithStrings/LazyEval.groovy**

```
quote = "Today ${-> company } stock closed at ${-> price }"

stocks.each { key, value ->
  company = key
  price = value
  println quote
}
```

这个简洁的版本会与上一版本产生同样的输出：

```
Today Apple stock closed at 663.01
Today Microsoft stock closed at 30.95
```

GString的惰性求值是个非常强大的概念。不过要小心，不要在字符串上栽跟头。如果希望改变表达式中使用的引用，而且希望它们的当前值被用于惰性求值中，请必须记住，不要在表达式中直接替换它们，而要使用一个无参闭包。

我们看到了Groovy的字符串操作和格式化的优雅，不过现在触及的只是一点皮毛而已。在Java中创建多行字符串既可怕又麻烦，下一节将介绍Groovy是如何简化这一过程的。

## 5.3　多行字符串

要在Java中创建一个多行字符串，我们不得不使用像str += ...这样的代码，使用+操作符连接多行，或者多次调用StringBuilder或StringBuffer的append()方法。

我们不得不使用大量的转义字符，这时候我们会抱怨："一定会有更好的方法。"Groovy就有。可以通过将字符串包含在3个单引号内（'''...'''）来定义多行字面常量，而且Groovy就是这样支持here文档[1]：

**WorkingWithStrings/MultilineStrings.groovy**

```
memo = '''Several of you raised concerns about long meetings.
To discuss this, we will be holding a 3 hour meeting starting
at 9AM tomorrow. All getting this memo are required to attend.
If you can't make it, please have a meeting with your manager to explain.
'''

println memo
```

下面是这段代码创建的多行字符串：

```
Several of you raised concerns about long meetings.
To discuss this, we will be holding a 3 hour meeting starting
at 9AM tomorrow. All getting this memo are required to attend.
If you can't make it, please have a meeting with your manager to explain.
```

就像可以使用双引号创建含有表达式的GString那样，也可以使用3个双引号创建包含表达式的多行字符串。

**WorkingWithStrings/MultilineStrings.groovy**

```
price = 251.12

message = """We're very pleased to announce
that our stock price hit a high of \$${price} per share
```

[1] here文档（here documents），又称作heredoc、hereis、here-字串或here-脚本，是一种在命令行shell（如sh、csh、ksh、bash、PowerShell和zsh）和程序语言（像Perl、PHP、Python和Ruby）里定义一个字串的方法。参见http://zh.wikipedia.org/wiki/Here%E6%96%87%E6%A1%A3。——译者注

```
on December 24th. Great news in time for...
"""
println message
```

Groovy会计算其中的表达式，如输出所示：

```
We're very pleased to announce
that our stock price hit a high of $251.12 per share
on December 24th. Great news in time for...
```

我每个月都要写份通讯。几年前，我决定将用于发送邮件通知的程序转换到Groovy。Groovy能够在多行字符串中嵌入一些值的能力正好派上用场。Groovy甚至使我写的通讯更容易被当作垃圾邮件了！（只是开玩笑，别当真。）

来看一个使用了这个特性的例子。假设有一个语言及其作者的映射，我们想为其创建一个XML表示。下面是一种实现方式：

**WorkingWithStrings/CreateXML.groovy**

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

content = ''
langs.each { language, author ->
  fragment = """
    <language name="${language}">
      <author>${author}</author>
    </language>
  """

  content += fragment
}
xml = "<languages>${content}</languages>"
println xml
```

这太可喜了，但是在17.1节我们将看到更令人惊叹的XML生成器。下面是使用多行字符串表达式产生的XML输出：

```
<languages>
  <language name="C++">
    <author>Stroustrup</author>
  </language>

  <language name="Java">
    <author>Gosling</author>
  </language>

  <language name="Lisp">
    <author>McCarthy</author>
  </language>
</languages>
```

这里使用嵌入了表达式的多行字符串创建了想要的内容，该内容是通过迭代包含数据的映射生成的。

看过了创建字符串的几种方式，下一节将探讨Groovy为操纵字符串提供的便捷方法。

## 5.4 字符串便捷方法

String的execute()方法的确好用，它帮我们创建了一个Process对象，所以只需要几行代码就可以执行系统级进程（参见2.1.3节）。

想要玩转String，还有其他可选方法。例如下面的代码，它使用了String的一个重载操作符：

**WorkingWithStrings/StringConvenience.groovy**

```
str = "It's a rainy day in Seattle"
println str

str -= "rainy "
println str
```

输出说明了这一重载的操作符的效果：

```
It's a rainy day in Seattle
It's a day in Seattle
```

-=操作符对于操纵字符串很有用，它会将左侧的字符串中与右侧字符串相匹配的部分去掉。Groovy在String类上添加的minus()方法使其成为可能（参见2.8节）。Groovy还向String类添加了其他便捷方法：plus()（+）、multiply()（*）、next()（++）、replaceAll()和tokenize()等。[1]

也可以在一系列String上迭代，如下所示：

**WorkingWithStrings/StringRange.groovy**

```
for(str in 'held'..'helm') {
  print "${str} "
}
println ""
```

这段代码生成的序列如下：

```
held hele helf helg helh heli helj helk hell helm
```

这里使用的仍然是java.lang.String。Groovy添加的所有这些设施可以帮助我们快速完成工作。

① http://groovy.codehaus.org/groovy-jdk/java/lang/String.html

现在我们知道了如何提取出部分字符串。核心的程序员往往也会接触到正则表达式，Groovy同样令事情更简单了，下一节即将介绍。

## 5.5　正则表达式

JDK包`java.util.regex`包含了使用正则表达式（RegEx）进行模式匹配的API。关于RegEx的详细讨论，请参考Jeffrey Friedl的*Mastering Regular Expressions*[Fri97]一书。别的方法不说，`String`类的`replaceFirst()`和`replaceAll()`方法就充分利用了RegEx模式匹配。为使编程使用RegEx更为容易，Groovy添加了一些操作符和符号。

~操作符可以方便地创建RegEx模式。这个操作符映射到`String`类的`negate()`方法：

**WorkingWithStrings/RegEx.groovy**

```
obj = ~"hello"
println obj.getClass().name
```

下面输出说明了所创建实例的类型：

```
java.util.regex.Pattern
```

前面的例子说明，将~应用于`String`，会创建一个`Pattern`实例。我们可以使用正斜杠、单引号或双引号来创建RegEx。正斜杠有一个优势：不必对反斜杠进行转义。因此，`/\d*\w*/`与`"\\d*\\w*"`等价，但是更优雅。

为方便匹配正则表达式，Groovy提供了一对操作符：`=~`和`==~`。下面就来探索一下这两个操作符之间的差别，以及它们的具体功能：

**WorkingWithStrings/RegEx.groovy**

```
pattern = ~"(G|g)roovy"
text = 'Groovy is Hip'
if (text =~ pattern)
  println "match"
else
  println "no match"

if (text ==~ pattern)
  println "match"
else
  println "no match"
```

运行这段代码，可以看出两个操作符之间的差别：

```
match
no match
```

`=~`执行RegEx部分匹配，而`==~`执行RegEx精确匹配。因此，在前面的代码示例中，第一个模式匹配报告的是match，而第二个报告的是no match。

=~操作符会返回一个Matcher对象，它是一个java.util.regex.Matcher实例。Groovy对Matcher的布尔求值处理不同于Java，只要至少有一个匹配，它就会返回true（参见2.7节）。如果有多个匹配，则matcher会包含一个匹配的数组。这有助于快速获得匹配给定RegEx的文本中的部分内容。

**WorkingWithStrings/RegEx.groovy**

```
matcher = 'Groovy is groovy' =~ /(G|g)roovy/
print "Size of matcher is ${matcher.size()} "
println "with elements ${matcher[0]} and ${matcher[1]}."
```

前面代码会报告Matcher的详细信息，具体如下：

```
Size of matcher is 2 with elements [Groovy, G] and [groovy, g].
```

可以使用replaceFirst()方法或replaceAll()方法方便地替换匹配的文本（顾名思义，前者仅替换第一个匹配，而后者会替换所有匹配）。

**WorkingWithStrings/RegEx.groovy**

```
str = 'Groovy is groovy, really groovy'
println str
result = (str =~ /groovy/).replaceAll('hip')
println result
```

原始的文本和替换后的文本分别如下：

```
Groovy is groovy, really groovy
Groovy is hip, really hip
```

作为总结，下面列出了与RegEx相关的Groovy操作符：

- 要从字符串创建一个模式，使用~操作符。
- 要定义一个RegEx，使用正斜杠，像/[G|g]roovy/中这样。
- 要确定是否存在匹配，使用=~。
- 对于精确匹配，使用==~。

这一章介绍了Groovy是如何将字符串的创建和使用变得比在Java中容易得多的。创建多行字符串和带表达式的字符串也是易如反掌。我们也看到Groovy如何简化了RegEx的使用。当着手使用普通字符串操作及正则表达式时，Groovy的字符串会让我们体会到改进。

在编程中，对象的集合也和处理字符串一样基础。Groovy利用闭包提供的便捷与流畅增强了JDK的集合类API，下一章我们将会看到。

第 6 章

# 使用集合类

6

编程时会经常使用集合类。Java开发包（JDK）有很多有用的集合类，Groovy扩展了它们，使它们用起来更方便了。例如，与传统的for循环相比，内部迭代器更简洁、更易用，而且出错的可能性更小。通常可以使用一个不同的专用迭代器find从集合中挑选出一个元素。如果要挑选出几个匹配的元素，只需要简单地把find改为findAll，剩下的代码还一样——这样很简洁，而且没有引入新的集合类时所要面对的负担。一旦熟悉了Groovy中的集合类，再回过头来使用Java的API可就难了。别说我没警告过你！

本章将使用JDK的集合类，不过学习的是Groovy提供的轻量级的、优美自然的方法。首先从List这种有序集合上的各种迭代器和便捷方法入手，之后再来看一下Map这种键值对关联集合上提供的类似方法。

## 6.1 使用 List

在Groovy中创建一个java.util.ArrayList实例要比在Java中容易。我们不必使用new或指明类名，而是可以简单地列出想包含在List中的初始值，如下所示：

**WorkingWithCollections/CreatingArrayList.groovy**

```
lst = [1, 3, 4, 1, 8, 9, 2, 6]
println lst
println lst.getClass().name
```

来看一下这个ArrayList的内容以及它的类型，Groovy的输出如下：

```
[1, 3, 4, 1, 8, 9, 2, 6]
java.util.ArrayList
```

在Groovy中声明一个列表时，引用lst指向的是一个java.util.ArrayList实例，如前面的输出所示。

[]操作符可用于获取List中的元素，如下面的例子所示：

**WorkingWithCollections/CreatingArrayList.groovy**

```
println lst[0]
println lst[lst.size() - 1]
```

输出显示了列表中的第一个元素和最后一个元素的值：

```
1
6
```

其实不必费那么大劲，跳好几个数去获得列表的最后一个元素，Groovy有更简单的方式。可以使用负的索引值，这时Groovy将从右侧开始遍历，而不是从左侧开始：

**WorkingWithCollections/CreatingArrayList.groovy**

```
println lst[-1]
println lst[-2]
```

这段代码也获得了该列表的最后两个元素，在输出中会看到：

```
6
2
```

甚至可以使用Range对象获得集合中的几个连续的值：

**WorkingWithCollections/CreatingArrayList.groovy**

```
println lst[2..5]
```

从位置2开始，列表中的4个连续的值如下：

```
[4, 1, 8, 9]
```

Range中还可以使用负的索引值，代码如下，其结果与前面代码相同：

**WorkingWithCollections/CreatingArrayList.groovy**

```
println lst[-6..-3]
```

快速地查看一下`lst[2..5]`实际返回的是什么：

**WorkingWithCollections/CreatingArrayList.groovy**

```
subLst = lst[2..5]
println subLst.dump()
subLst[0] = 55
println "After  subLst[0]=55 lst = $lst"
```

我们会看到`dump()`方法所报告的该实例的详细信息，以及修改之后的列表：

```
<java.util.ArrayList$SubList@fedbf parent=[1, 3, 4, 1, 8, 9, 2, 6]
  parentOffset=2 offset=2 size=4 this$0=[1, 3, 4, 1, 8, 9, 2, 6] modCount=1>
After  subLst[0]=55 lst = [1, 3, 55, 1, 8, 9, 2, 6]
```

如果使用一个像`2..5`这样的`Range`作为索引，`java.util.ArrayList`会返回一个指向原来列表部分内容的实例[1]。所以请注意，得到的并非副本，如果修改了其中一个列表的元素，另一个也会受到影响。

可以看到Groovy是如何让`List`的应用编程接口（Application Programming Interface，API）变得更简单的。我们使用的是同样的、存在已久的`ArrayList`，但是当以Groovy之角度去看时，它漂亮和轻便了许多。

Groovy提供的便利远不止创建列表这么一点，下一节将介绍更多内容。

## 6.2　迭代 ArrayList

我们经常要对一组值进行导航或迭代。Groovy提供了优雅的列表迭代方式，还支持迭代时在其中的值上优雅地执行操作。

### 6.2.1　使用List的each方法

第4章中介绍过Groovy为迭代集合提供的便捷方式。这个迭代器（以`each()`命名的方法）也称内部迭代器。

**内部迭代器与外部迭代器**

我们习惯于C++和Java等语言中的外部迭代器，它们能让其用户或客户来控制迭代。我们必须检查迭代是不是要结束了，并且需要显式地移动到下一个元素。

内部迭代器在支持闭包的语言中很流行，迭代器的用户或客户不控制迭代，他们只需要提供一个要在集合中的每个元素上执行的代码块。

内部迭代器更容易使用，我们不必控制迭代。外部迭代器则更为灵活，我们可以更方便地控制迭代顺序、跳过元素、结束迭代或重新迭代等因素。

不要让缺乏灵活性这种表象阻碍我们使用内部迭代器。为了提供更多的灵活性和便捷性，内部迭代器的实现者往往需要付出额外的努力。以`List`为例，要控制迭代的灵活性，这要通过本章将要介绍的不同便捷方法的形式来实现。

下面创建一个`ArrayList`，然后使用`each()`方法对它进行迭代。

[1] 在较新的Groovy版本中，这种行为有所改变，所以读者一定要明确所使用的版本，具体信息请参考相应文档。——译者注

**WorkingWithCollections/IteratingArrayList.groovy**

```
lst = [1, 3, 4, 1, 8, 9, 2, 6]

lst.each { println it }
```

在迭代时打印出每个元素，输出如下：

```
1
3
4
1
8
9
2
6
```

可以使用reverseEach()方法反向迭代元素。如果关注迭代过程中的计数或索引，可以使用eachWithIndex()方法。

还可以执行其他操作，比如对闭包中的元素求和（参见4.3节），如下所示：

**WorkingWithCollections/IteratingArrayList.groovy**

```
total = 0
lst.each { total += it }
println "Total is $total"
```

下面是这段代码的执行结果：

```
Total is 34
```

假设想把集合中的每个元素变为原来的2倍，可以使用each()方法试试：

**WorkingWithCollections/IteratingArrayList.groovy**

```
doubled = []
lst.each { doubled << it * 2 }

println doubled
```

结果如下：

```
[2, 6, 8, 2, 16, 18, 4, 12]
```

这里创建了一个名为doubled的空ArrayList来保存结果。在对集合进行迭代时，将每个元素加倍，并使用<<操作符（映射到leftShift()方法）将所得的值放到结果中。

如果想在集合中的每个元素上执行一些操作，each()方法是一个不错的选择，但如果想让这些操作生成一些结果，则可以求助其他方法。

### 6.2.2　使用List的collect方法

如果想在集合中的每个元素上执行操作并返回一个结果集合，Groovy提供了一个简单的解决方案——collect()方法，下面我们会看到：

**WorkingWithCollections/IteratingArrayList.groovy**

```
println lst.collect { it * 2 }
```

collect()方法和each()一样，会在集合中的每个元素上调用传入的闭包。不过它会把闭包的返回值收集（即collect一词的本意）到一个集合中，最后返回这个生成的结果集合。前面例子中的闭包返回给定值的2倍，因为闭包有一个隐式的return。得到了一个ArrayList，其中的元素值是输入值的2倍，在输出中会看到：

```
[2, 6, 8, 2, 16, 18, 4, 12]
```

如果想在集合的每个元素上执行操作，可以使用each()。然而，如果想得到一个进行了这类计算的结果集合，则要使用collect()方法。

可用的还不止这两个内部迭代器，请看下节。

## 6.3　使用查找方法

我们知道了如何在集合上执行迭代，以及如何在每个元素上执行操作。然而，如果想搜索特定的元素，each()和collect()都不方便。此时应该使用find()，像下面这样：

**WorkingWithCollections/Find.groovy**

```
lst = [4, 3, 1, 2, 4, 1, 8, 9, 2, 6]

println lst.find { it == 2 }
```

这段代码会挑出集合中第一个等于2的元素，在输出中我们会看到：

```
2
```

这段代码会查找一个集合中与2这个值匹配的对象，find()会找到第一次出现的匹配对象。在这种情况下，它返回的是在位置3的对象。就像each()方法一样，find()方法也会对集合进行迭代，但是它只会迭代到闭包返回true为止。一得到true，find()就会停止迭代，并将当前的元素返回。如果遍历结束也没得到true，则返回null。

我们可以在附到find()上的闭包中指定任何条件。例如，下面是如何查找第一个大于4的元素：

**WorkingWithCollections/Find.groovy**

```
println lst.find { it > 4 }
```

这段代码会报告列表中第一个大于4的数：

```
8
```

也可以找到列表中出现的所有2。就像find()方法的表现类似each()一样，findAll()方法的表现类似collect()：

WorkingWithCollections/Find.groovy

```
println lst.findAll { it == 2 }
```

可以看到在该列表中找到的所有的2：

```
[2, 2]
```

在这个例子中，要查找2，findAll()方法返回的是对象，而不是位置。如果想查找第一个匹配对象的位置，可以使用findIndexOf()方法。

在最简单的情况下，找到所有的2听上去并不是很有用。然而，一般而言，如果正在查找匹配某些标准的对象，我们就会得到那些对象。例如，如果要查找人口数大于某个特定值的所有城市，结果就会是相应城市的列表。回到前面的例子，如果想要所有大于4的数，应该这样做：

WorkingWithCollections/Find.groovy

```
println lst.findAll { it > 4 }
```

这段代码会报告所有大于4的元素：

```
[8, 9, 6]
```

本节介绍了如何迭代集合以及从集合中选择元素。因为对集合的操作远不止这么几个，所以Groovy使用更多便捷方法扩展了其流畅性，下面我们会看到。

## 6.4 List 上的其他便捷方法

Groovy向集合类Collection添加了很多便捷方法。（具体列表请参考http://groovy.codehaus.org/groovy-jdk/java/util/Collection.html。）下面使用我们已经熟悉的方法each()来实现一个例子。之后再使用那些可以使代码自包含、表现力更好的方法来重构这个例子。在此过程中，我们将会看到Groovy是如何像函数式编程语言那样将代码块看作一等公民的。

假设有一个字符串集合，然后想计算其中总的字符数。下面是使用each()方法实现的一种方式：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
lst = ['Programming', 'In', 'Groovy']

count = 0
```

```
lst.each { count += it.size() }
println count
```

求出的字符数目如下：

```
19
```

Groovy往往会为一个任务提供多种解决方式。下面是另一种方式，使用了collect()和sum()（都是Groovy在JDK的集合类上添加的方法）：

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
println lst.collect { it.size() }.sum()
```

在collect()方法返回的集合上调用了sum()方法，这段代码会产生与前一版本同样的输出：

```
19
```

代码有点简洁，但却是自包含的：要处理集合中每个单独的元素，而且要获得一个累积的结果，each()很有用。然而，如果想在集合的每个元素上执行一些计算，同时把结果保留为一个集合，collect()就很有用了。这一点也可以应用于在集合上进行遍历计算的其他操作（比如sum()方法）上。

使用inject()方法可以实现同样的功能：

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
println lst.inject(0) { carryOver, element ->  carryOver + element.size() }
```

输出如下：

```
19
```

inject()会对集合中的每个元素调用闭包。在这个例子中，集合中的元素是用参数element表示的。inject()会把将要注入的一个初始值当做一个参数，并通过carryOver参数把它放到第一次对闭包的调用中。之后它会把从闭包获得的结果注入到随后对闭包的调用中。如果想在集合中的每个元素上应用某个计算，获得一个累积的结果，与collect()方法相比，我们会首选inject()方法。

假设想把集合中的元素连接成一个句子。利用join()可以很方便地实现：

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
println lst.join(' ')
```

下面是连接元素的结果：

```
Programming In Groovy
```

join()会迭代每个元素，然后将每个元素和作为输入参数给定的字符连接起来。在这个例子

中，作为输入参数给定的是空格字符，因此join()方法返回了字符串Programming In Groovy。当我们想连接一个由路径组成的集合时，join()方法就派上用场了，例如使用一个分号（:）构造一个classpath，一个简单的调用即可完成。

可以通过索引替换List中的一个元素。下面的代码将['Be', 'Productive']设置给了元素0：

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
lst[0] = ['Be', 'Productive']
println lst
```

这会导致集合中包含一个List，就像下面这样：

```
[[Be, Productive], In, Groovy]
```

如果这不是我们想要的，可以使用flatten()方法将List拉平：

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
lst = lst.flatten()
println lst
```

结果是一个拉平了的单一列表：

```
[Be, Productive, In, Groovy]
```

还可以在List上使用-操作符（minus()）方法，像这样：

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
println lst - ['Productive', 'In']
```

右操作数中的元素会被从左侧的集合中移除。如果提供了一个不存在的元素，不用担心——它会被直接忽略掉。-操作符很灵活，可以为右操作数提供一个列表或是单个的值。该操作的结果如下：

```
[Be, Groovy]
```

可以使用reverse()方法得到列表的一份副本，其中的元素是反向排列的。

下面是Groovy中的另一个便捷方法：可以很方便地在每个元素上执行操作，而不用显式地使用迭代。

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
println lst.size()
println lst*.size()
```

这段代码会打印元素个数，以及每个元素的大小：

```
4
[2, 10, 2, 6]
```

6

第一次调用size()是在列表上，它会返回4，即列表中当前的元素个数。第二次调用，因为*的影响，即作用于列表中的每个元素（这个例子中是String）的展开操作符（spread operator）的影响，会返回一个List，其中的每个元素分别保存原始集合中相应元素的大小。lst*.size()的作用与lst.collect { it.size() }相同。

最后来看一下如何在方法调用中使用ArrayList。如果一个方法接收很多参数，不同于发送单个的参数，我们可以将一个ArrayList打散作为参数；也就是说，使用*操作符将集合拆成单个对象，下面我们将看到。要使其正常工作，ArrayList的元素个数必须与方法期望的参数个数相同。

**WorkingWithCollections/CollectionsConvenienceMethods.groovy**

```
def words(a, b, c, d) {
  println "$a $b $c $d"
}

words(*lst)
```

下面是使用展开操作符的结果：

```
Be Productive In Groovy
```

本节探索了Groovy用于处理由对象组成的List的设施，下一节将介绍在Groovy中如何使用Map。

## 6.5　使用 Map 类

Java的java.util.Map在使用键值对的关联集合时很有用。通过使用闭包，Groovy使Map用起来更为简单和优雅。创建一个Map实例也很简单，因为不需要使用new或指明任何类名。只需简单地创建键值对即可：

**WorkingWithCollections/UsingMap.groovy**

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

println langs.getClass().name
```

通过输出确认一下所创建集合的类名：

```
java.util.LinkedHashMap
```

这个例子创建了一个散列映射（hash map），以一些编程语言为键，以语言相应的作者为值。键和值用冒号（:）分隔，整个映射放在一个中括号（[]）中。这种简单的Groovy语法创建了一个java.util.LinkedHashMap实例。通过调用getClass()并获得其name属性便可看到。为什么要这么啰嗦地调用getClass()方法，而没有使用JavaBean的约定直接访问class属性？这里有一

个小陷阱。

可以使用[]操作符来访问一个键的值，如下面代码所示：

**WorkingWithCollections/UsingMap.groovy**

```
println langs['Java']
println langs['C++']
```

下面是所请求的两个键的值：

```
Gosling
Stroustrup
```

如果想看点花哨的，Groovy当然不会让我们失望。可以把键用作好像是Map的一个属性，以此访问该键对应的值：

**WorkingWithCollections/UsingMap.groovy**

```
println langs.Java
```

Groovy会将该键用作一个属性，返回相应的值：

```
Gosling
```

这很巧妙，把键看作好像是对象的一个属性，然后发送该键，很是方便，而且Map会聪明地返回其值。当然，有经验的程序员立即会问："有什么陷阱？"这个陷阱已经很明显了：我们不能在Map上调用class属性，Map会假定class这个名字指向的是一个（不存在的）键，而返回null。很明显，后面再调用class的name属性，因为是在null上调用，就会失败了。当调用class属性时，Map和其他一些类的实例不会返回Class元对象。为避免结果出乎意料，在实例上要总是使用getClass()方法，而不是class属性。

所以必须调用getClass()方法。但是C++这个键又怎么样呢？我们来试一下：

**WorkingWithCollections/UsingMap.groovy**

```
println langs.C++ // 不合法代码
```

下面是我们得到的输出：

```
java.lang.NullPointerException: Cannot invoke method next() on null object
```

这是什么意思？我们可能会放弃这个代码示例，然后说C++不管在哪都要出问题。

但是，这个问题实际上缘自与Groovy的另一个特性——操作符重载——的冲突（参见2.8节）。Groovy会把前面的请求看作要获取键C的值，而这个键不存在。因此，它会返回null，然后尝试调用++操作符所映射的next()方法。幸运的是，像这样的特殊情况有一个变通方案：只需把带有这种会惹麻烦的字符的键值看作一个String。

**WorkingWithCollections/UsingMap.groovy**

```
println langs.'C++'
```

太棒了！终于得到了正确的输出：

```
Stroustrup
```

Groovy添加了另一个创建Map的便捷方法。当定义一个Map时，对于规规矩矩的键名，可以省略其引号。例如，我们不使用键上的引号，重写编程语言和它们的作者这个Map：

**WorkingWithCollections/UsingMap.groovy**

```
langs = ['C++' : 'Stroustrup', Java : 'Gosling', Lisp : 'McCarthy']
```

我们知道了如何创建一个Map，以及如何访问这种集合中的单个值。下一节将介绍如何在Map这种集合上迭代。

## 6.6 在 Map 上迭代

Groovy向Map添加了很多便捷方法[①]。我们可以在一个Map上迭代，就像在ArrayList上迭代那样（参见6.2节）。

Map也有自己的each()和collect()方法。

### 6.6.1 Map的each方法

来看一个使用each()方法的例子：

**WorkingWithCollections/NavigatingMap.groovy**

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

langs.each { entry ->
  println "Language $entry.key was authored by $entry.value"
}
```

输出如下：

```
Language C++ was authored by Stroustrup
Language Java was authored by Gosling
Language Lisp was authored by McCarthy
```

如果附到each()上的闭包只接收一个参数，each()会把一个MapEntry实例发送给该参数。如果想单独获得键和值，只需要在闭包中提供两个参数，如下面的例子所示：

① http://groovy.codehaus.org/groovy-jdk/java/util/Map.html

WorkingWithCollections/NavigatingMap.groovy

```
langs.each { language, author ->
  println "Language $language was authored by $author"
}
```

下面是使用两个参数的闭包在Map上迭代的输出：

```
Language C++ was authored by Stroustrup
Language Java was authored by Gosling
Language Lisp was authored by McCarthy
```

这个代码示例使用each()方法在langs集合上迭代，迭代时会使用一个键和一个值调用用闭包。在闭包中，我们分别使用变量名language和author引用这两个参数。

类似地，对于其他方法，比如collect()、find()等，如果只想要MapEntry，就使用一个参数；如果想分别获得键和值，则使用两个参数。

### 6.6.2 Map的collect方法

下面试试Map中的collect()方法。首先，它与ArrayList中的collect()方法类似的是，都返回一个列表。不过，如果想让Map的collect()向我们的闭包发送一个MapEntry，就定义一个参数；否则就定义两个参数，分别表示键和值，如下所示：

WorkingWithCollections/NavigatingMap.groovy

```
println langs.collect { language, author ->
  language.replaceAll("[+]", "P")
}
```

代码返回如下的列表：

```
[CPP, Java, Lisp]
```

在前面的代码中，我们创建了一个键的列表，键中出现的所有+都替换成了字符P。

我们可以方便地将Map中的数据转换成其他表示形式。例如，在17.1节我们会看到创建一个XML表示是多么容易。

### 6.6.3 Map的find和findAll方法

Groovy也向Map添加了find()和findAll()方法。我们看一个例子：

WorkingWithCollections/NavigatingMap.groovy

```
println "Looking for the first language with name greater than 3 characters"
entry = langs.find { language, author ->
  language.size() > 3
```

6

```
}
println "Found $entry.key written by $entry.value"
```

使用find()方法的输出如下：

```
Looking for the first language with name greater than 3 characters
Found Java written by Gosling
```

find()方法接收一个闭包，而该闭包接收键和值（再次强调，要接收MapEntry则使用一个参数）。与ArrayList中的对应方法类似，如果闭包返回true，它会退出迭代。在前面的示例代码中，我们会找到名字多于3个字符的第一门语言。如果直到最后闭包也没有返回true，该方法会返回null。否则，它会返回Map中一个匹配条目的实例。

可以使用findAll()方法获得匹配所查找条件的所有元素，如下面例子所示：

**WorkingWithCollections/NavigatingMap.groovy**

```
println "Looking for all languages with name greater than 3 characters"
selected = langs.findAll { language, author ->
  language.size() > 3
}
selected.each { key, value ->
  println "Found $key written by $value"
}
```

这段代码会报告所有满足给定条件的语言：

```
Looking for all languages with name greater than 3 characters
Found Java written by Gosling
Found Lisp written by McCarthy
```

除了内部迭代器，Groovy还提供了强大的便捷函数，对Map中的元素进行选择和分组，下面我们会看到。

## 6.7 Map 上的其他便捷方法

我们来看一些Map上的便捷方法，并以此结束关于集合的讨论。

要取得满足某个给定条件的一个元素，find()方法很有用。然而，如果不是想得到元素，而只是想确定集合中是否有任何元素满足某些条件，就要用any()方法。

继续使用6.6节中的语言及其作者的例子。可以使用any()方法来确定是否在有些语言的名字中包含着非字母的字符：

**WorkingWithCollections/NavigatingMap.groovy**

```
print "Does any language name have a nonalphabetic character? "
println langs.any { language, author ->
  language =~ "[^A-Za-z]"
}
```

因为键中包含C++，所以代码会报告如下内容：

```
Does any language name have a nonalphabetic character? true
```

就像之前所讨论的Map上的其他方法一样，any()也接受了一个带2个参数的闭包。这个示例中的闭包使用了一个正则表达式比较（参见5.5节），来确定是否有语言的名字中包含非字母的字符。

any()方法会查找至少一个满足给定条件（谓词）的Map中的元素，而every()方法则会检查是否所有的元素都满足给定条件：

**WorkingWithCollections/NavigatingMap.groovy**

```
print "Do all language names have a nonalphabetic character? "
println langs.every { language, author ->
  language =~ "[^A-Za-z]"
}
```

输出会告诉我们，是否所有元素都满足给定条件：

```
Do all language names have a nonalphabetic character? false
```

如果想基于某些标准对Map中的元素进行分组，不必费劲地执行迭代或循环，groupBy()就是干这个的。要做的只是通过闭包说明判断标准。这里有一个例子：friends指向的是一个包含一些朋友的Map，其中很多人名字（First Name，不包括姓）相同。如果想根据名字对朋友进行分组，只需要调用一个groupBy()即可完成，如下面的代码所示。在附到groupBy()之后的闭包中，指明了所要进行的分组——在这个例子中，从全名（Full Name）中取出名字并返回。一般而言，会按我们感兴趣的分类返回属性。例如，如果使用属性firstName和lastName将朋友的名字保存在一个Person对象中，而不是使用一个简单的String，我们可以将闭包写成{ it.firstName }。在下面的代码中，groupByFirstname是一个Map，其中名字作为键，而键相应的值本身又是一个名字和相应全名组成的Map。最后，我们对结果进行迭代，并打印出值：



**WorkingWithCollections/NavigatingMap.groovy**

```
friends = [ briang : 'Brian Goetz', brians : 'Brian Sletten',
        davidb : 'David Bock', davidg : 'David Geary',
        scottd : 'Scott Davis', scottl : 'Scott Leberknight',
        stuarth : 'Stuart Halloway']

groupByFirstName = friends.groupBy { it.value.split(' ')[0] }

groupByFirstName.each { firstName, buddies ->
  println "$firstName : ${buddies.collect { key, fullName -> fullName }.join(', ')}"
}
```

下面是每个组中的结果：

```
Brian : Brian Goetz, Brian Sletten
David : David Bock, David Geary
Scott : Scott Davis, Scott Leberknight
Stuart : Stuart Halloway
```

最后还要记住两个技巧：Groovy将`Map`用于具名参数（参见2.3节），以及使用`Map`实现接口（参见2.6节）。

在本章中，我们看到了将闭包引入到Java的集合API中的强大。随着将这些概念应用于项目，我们将发现，使用集合类更容易、更快速，代码也会更简短，这很有趣。你还会发现，使用Groovy编程，会让在其他语言中看似平常的遍历和操纵集合这种任务都充满了激情。

看过了Groovy语言的能力以及这门语言向不同API注入的流畅性，我们已经为进阶打好了基础。下一部分，将介绍如何将这门语言有效地应用于诸如处理XML和访问数据库等操作。

*Part 2*

第二部分

# 使用 Groovy

本部分内容

# 第 7 章 探索GDK

Groovy不仅给Java虚拟机（JVM）带来了动态语言的优势，还增强了存在已久的Java开发包（JDK）的性能。因为享受得到更好、更轻量级、更优雅的Java API，使用Groovy编程的效率很高。

前面已经介绍过Groovy通过便捷方法对JDK的增强，其中很多方法大量使用了闭包。Groovy的这一扩展称作Groovy Java开发包（Groovy Java Development Kit，Groovy JDK）或GDK。[①]

图7-1显示了JDK和GDK之间的关系。GDK是基于JDK的，所以在Java代码和Groovy代码之间传递对象时，无需任何转换。当处在同一JVM中时，Java端和Groovy端使用的是同一对象。对于Groovy端看到的对象，因为Groovy向其中添加了便于使用、可以提高开发效率的方法，所以看上去更时髦。




图7-1　JDK和GDK

GDK扩展了一些JDK中的类，我们会在本书的不同章节中讨论其中的一部分。本章主要关注两个方面：一个是对`java.lang.Object`类的扩展，另一个是对常用类的其他各种扩展。

## 7.1　使用 Object 类的扩展

本节将探索Groovy对众类之母`java.lang.Object`的一些扩展。在第6章中，我们看到了Groovy在`Collection`上添加的方法：`each()`、`collect()`、`find()`、`findAll()`、`any()`和`every()`。其实不仅`Collection`上提供了这些方法，我们在任何对象上都可以使用它们。这为

① http://groovy.codehaus.org/groovy-jdk

我们以类似方式使用不同对象和集合类提供了一致的API——这是组合模式的优点之一（参见*Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]）。Groovy还在`Object`上添加了一些和集合类无关的便捷方法。我们无意将本章变成GDK库的完整参考手册，因此这里暂不介绍所有这些方法。相反，重点将放在那些有可能激发我们的兴趣，或者对日常任务很有帮助的方法上。

### 7.1.1　使用dump和inspect方法

如果想知道类的一个实例包含哪些内容，可以在运行时使用`dump()`方法轻松获得：

**ExploringGDK/ObjectExtensions.groovy**

```
str = 'hello'

println str
println str.dump()
```

看一下这段代码打印的该对象的详细信息：

```
hello
<java.lang.String@5e918d2 value=hello offset=0 count=5 hash=99162322>
```

`dump()`使我们得以一窥对象内部。我们可以将其用于调试、日志和学习。它会给出目标实例的类型（class）、散列码及字段。

Groovy还向`Object`类添加了另一个方法——`inspect()`。该方法旨在说明创建一个对象需要提供什么。如果类没有实现该方法，会简单地返回`toString()`所返回的内容。如果对象要接收大量输入，该方法可以帮助类的使用者在运行时确定他们应该提供的内容。

### 7.1.2　使用上下文with()方法

JavaScript和VBScript都有`with`这个友好的特性，支持创建一个上下文（context）。在`with`的作用域内调用的任何方法，都会被定向到该上下文对象，这样就去掉了对该实例的多余引用。在Groovy中，`Object`的`with()`方法提供了同样功能。（Groovy中`with()`方法是作为`identity()`的同义词引入的，所以它们可以互换使用。）该方法接受一个闭包作为参数。在闭包内调用的任何方法都会被自动解析到上下文对象。我们看一个例子，先从没有利用这种简洁性的代码开始：

**ExploringGDK/Identity.groovy**

```
lst = [1, 2]
lst.add(3)
lst.add(4)
println lst.size()
println lst.contains(2)
```

7

在前面的代码中，我们一直在调用lst上的方法，其中lst引用的是一个ArrayList实例。这里没有隐含的上下文，我们重复地（或者说冗余地）使用了对象引用lst。在Groovy中，可以使用with()方法设置一个上下文，因此代码可以改成下面这样：

**ExploringGDK/Identity.groovy**

```
lst = [1, 2]
lst.with {
  add(3)
  add(4)
  println size()
  println contains(2)
}
```

这段代码噪音很少，其输出如下：

```
4
true
```

with()方法是怎样知道把闭包内的调用路由到上下文对象的呢？魔力在于该闭包的delegate属性（更多信息，参见4.9节）。我们检查一下附到with()上的闭包中的delegate属性、this属性以及owner属性。

**ExploringGDK/Identity.groovy**

```
lst.with {
  println "this is ${this},"
  println "owner is ${owner},"
  println "delegate is ${delegate}."
}
```

输出说明了我们所关注的引用的详细信息：

```
this is Identity@ce56f8,
owner is Identity@ce56f8,
delegate is [1, 2, 3, 4].
```

当我们调用with()方法时，它会将该闭包的delegate属性设置到调用with()的对象上。正如4.9节所讨论的，delegate会负责this不处理的方法。

with()方法使我们更方便地在一个对象上调用多个方法，利用上下文来减少混乱吧。在构建领域特定语言（DSL）时，会发现该方法非常有用。还可以实现类似脚本的调用，隐式地将其路由到幕后的对象，第19章中将详述。

### 7.1.3　使用sleep

添加到Object上的sleep()方法应该叫作酣睡（soundSleep），因为在给定的毫秒数时间（近似）内睡眠时，该方法会忽略中断。

我们看一个使用sleep()方法的例子：

ExploringGDK/Sleep.groovy

```
thread = Thread.start {
  println "Thread started"
  startTime = System.nanoTime()
  new Object().sleep(2000)
  endTime = System.nanoTime()
  println "Thread done in ${(endTime - startTime)/10**9} seconds"
}
new Object().sleep(100)
println "Let's interrupt that thread"
thread.interrupt()
thread.join()
```

输出说明，该线程忽略了中断，并完成执行：

```
Thread started
Let's interrupt that thread
Thread done in 2.000272 seconds
```

这里我们使用了Groovy添加的Thread.start()方法。这是在一个不同的线程中执行一段代码的方便方法。在Object上调用sleep()与使用 Java提供的Thread.sleep()的区别在于：如果有InterruptedException，前者会压制下来。如果我们确实想被中断，也不必受try-catch之苦。相反，可以在前面的sleep()方法上使用一个变种版本，它接受一个处理中断的闭包：

ExploringGDK/Sleep.groovy

```
def playWithSleep(flag)
{
  thread = Thread.start {
    println "Thread started"
    startTime = System.nanoTime()
    new Object().sleep(2000) {
      println "Interrupted... " + it
      flag
    }
    endTime = System.nanoTime()
    println "Thread done in ${(endTime - startTime)/10**9} seconds"
  }

  thread.interrupt()
  thread.join()
}

playWithSleep(true)
playWithSleep(false)
```

在输出中可以看到闭包是如何处理中断的：

```
Thread started
Interrupted... java.lang.InterruptedException: sleep interrupted
Thread done in 0.00437 seconds
Thread started
Interrupted... java.lang.InterruptedException: sleep interrupted
Thread done in 1.999077 seconds
```

在中断处理器内，我们可以采取任何适当的动作。如果需要访问InterruptedException，也可以，它作为闭包的一个参数存在。如果我们在闭包内返回一个false值，sleep()将继续，就好像没有被中断，在前面代码中，通过第二个playWithSleep()调用语句可以看到这一点。

## 7.1.4　间接访问属性

我们知道，Groovy使访问属性变得非常容易。例如，对于一个Car类型的car实例，要获得其miles属性，可以简单地调用car.miles。然而，如果编写代码时不知道属性名，这种语法就派不上用场了，比如属性名依赖于用户的输入，而且我们不想为所有可能的输入硬编码一个处理分支。这时可以使用[]操作符（该操作符映射到Groovy添加的getAt()方法）动态地访问属性。如果将该操作符用于赋值语句的左侧，它则映射到putAt()方法。

我们看一个例子：

**ExploringGDK/IndirectProperty.groovy**

```
class Car {
  int miles, fuelLevel
}

car = new Car(fuelLevel: 80, miles: 25)

properties = ['miles', 'fuelLevel']
// 上面的列表可能通过一些输入来填充
// 或者来自一个Web应用中的动态表单

properties.each { name ->
  println "$name = ${car[name]}"
}

car[properties[1]] = 100

println "fuelLevel now is ${car.fuelLevel}"
```

我们能够间接地与该实例交互，如输出所示：

```
miles = 25
fuelLevel = 80
fuelLevel now is 100
```

这就是使用[]操作符访问miles和fuelLevel属性的过程。这种方法可应用于通过输入接收

的属性名，例如动态创建和填充Web表单。可以轻松地编写一个高阶函数，让它接受属性名列表和一个实例，并以XML、HTML或任何其他我们期望的格式来输出这些属性名和它们的值。可以通过对象的properties属性（也就是getProperties()方法）获得其所有属性的列表。

### 7.1.5　间接调用方法

如果以String形式接收到方法名，而且想调用该方法，使用反射要这样实现——首先必须从实例取到Class元对象，然后调用getMethod()方法得到Method实例，最后在该实例上调用invoke()方法。噢，还有，别忘了那些不得不处理的异常。

在Groovy中不需要做这些，而只需简单地调用invokeMethod()方法。Groovy中的所有对象都支持该方法。这是一个例子：

**ExploringGDK/IndirectMethod.groovy**

```
class Person {
  def walk() { println "Walking..." }
  def walk(int miles) { println "Walking $miles miles..." }
  def walk(int miles, String where) { println "Walking $miles miles $where..." }
}

peter = new Person()

peter.invokeMethod("walk", null)
peter.invokeMethod("walk", 10)
peter.invokeMethod("walk", [2, 'uphill'] as Object[])
```

下面是间接调用该方法的输出：

```
Walking...
Walking 10 miles...
Walking 2 miles uphill...
```

因此，如果编写代码时不知道方法名，而在运行时获得，那就可以使用几行代码将其变为实例上的动态调用。

Groovy还提供了getMetaClass()方法，用以获得元类（metaclass）对象，这是在Groovy中利用动态能力的一个关键对象，在后面的章节中我们将看到。

Groovy的扩展API远不止扩展了JDK中最基础的Object类，下面我们会看到。

## 7.2　其他扩展

GDK所做的扩展远不止Object类，其他一些JDK的类和接口也得到了增强。再次重申，GDK的扩展很广，本节所涉及的只是一个子集。这里要介绍的是那些最常用的扩展。

### 7.2.1　数组的扩展

在所有数组类型上，比如int[]、double[]和char[]等，都可以使用Range对象作为索引（创建数组的语法请参见2.11.7节）。下面演示了如何使用索引的范围访问一个int数组中的连续几个值：

ExploringGDK/Array.groovy

```
int[] arr = [1, 2, 3, 4, 5, 6]

println arr[2..4]
```

输出显示了给定范围内的值：

```
[3, 4, 5]
```

GDK向List、Collection和Map添加了很多便捷方法，通过第6章的学习我们已经很熟悉了。

### 7.2.2　使用java.lang的扩展

对于基本类型的包装器，如Character、Integer等，有一个值得注意的补充，那就是重载操作符映射的方法，比如+操作符映射的plus()、++操作符映射的next()等。当创建DSL时，我们会发现这些方法（或者应该说是操作符）很有用。

Number（Integer和Double就扩展了该类）加上了迭代器方法upto()和downto()。它还有step()方法（参见2.1.2节）。这些方法有助于对一个范围内的值进行迭代。

在2.1.3节中，我们看了一些与系统级进程交互的例子。Process类提供了访问stdin、stdout和stderr命令的便捷方法，分别对应out、in和err属性。它还有一个text属性，可以为我们提供完整的标准输出或来自进程的响应。如果想一次性读取完整的标准错误，可以在进程实例上使用err.text。使用<<操作符可以以管道方式链接到一个进程中。（管道——|，在类Unix系统上用于将一个进程的输出链接到另一个进程的输入。）下面通过一个例子来看一下与一个wc进程的通信——wc程序是类Unix系统上一个流行的实用程序，它会向标准输出打印从其标准输入中发现的单词数、行数和字符数：

ExploringGDK/UsingProcess.groovy

```
process = "wc".execute()

process.out.withWriter {
  // 将输入发送到进程
  it << "Let the World know...\n"
  it << "Groovy Rocks!\n"
}
```

```
// 从进程读取输入
println process.in.text
// 或
//println process.text
```

前面代码的输出是wc返回的结果——2行，6个单词，36个字符：

```
2       6       36
```

在这段代码中，首先，通过调用String的execute()获得了一个进程实例。我们想向wc的标准输入写内容，所以需要来自程序的一个OutputStream。可以通过调用out属性从进程获取。

要写入内容，可以使用<<操作符。然而，一旦把数据写到了流中，我们就想刷新（flush）并关闭该流。可以使用一个方法——withWriter()，同时完成这两方面的处理。该方法会将OutputStreamWriter附到OutputStream上，同时将其传给闭包。当我们从闭包返回时，它会自动刷新并关闭流（参见4.5节）。

试试用Java实现前面这段代码，你就能体会到，Groovy不仅节省时间，还带来了优雅的享受。

如果想将命令行参数发送给进程，有两个选择：把参数格式化为一个字符串，或者创建一个参数的String数组。String[]也支持execute()方法，数组的第一个元素会被当作要执行的命令，其余元素则被视作该命令的参数。作为替代，可以使用List的execute()方法。

这是一个向groovy命令传递命令行参数的例子：

**ExploringGDK/ProcessParameters.groovy**

```
String[] command = ['groovy', '-e', '"print \'Groovy\'"']
println  "Calling ${command.join(' ')}"
println command.execute().text
```

上述代码执行的命令及输出如下：

```
Calling groovy -e "print 'Groovy'"
Groovy
```

在Groovy中，我们可以非常轻松地启动一个进程、发送参数，以及与该进程交互。只需要几行代码。

如果想创建多个线程，并将任务分派给单独的线程执行，Groovy可以让我们少敲很多字。使用start()方法启动一个线程，并为其提供一个将在单独的线程中执行的闭包。如果希望该线程是守护线程（daemon thread），可以使用startDaemon()方法代替。如果当前已经没有活跃的非守护线程，守护线程会退出——有点像只有老板在的时候才干活的员工。以下是这两个方法的实际演示例子：

**ExploringGDK/ThreadStart.groovy**

```
def printThreadInfo(msg) {
```

```
  def currentThread = Thread.currentThread()
  println "$msg Thread is ${currentThread}. Daemon? ${currentThread.isDaemon()}"
}

printThreadInfo 'Main'

Thread.start {
  printThreadInfo "Started"
  sleep(3000) { println "Interrupted" }
  println "Finished Started"
}

sleep(1000)

Thread.startDaemon {
  printThreadInfo "Started Daemon"
  sleep(5000) { println "Interrupted" }
  println "Finished Started Daemon" // 不会执行到这里
}
```

下面的输出说明了线程信息：

```
Main Thread is Thread[main,5,main]. Daemon? false
Started Thread is Thread[Thread-1,5,main]. Daemon? false
Started Daemon Thread is Thread[Thread-2,5,main]. Daemon? true
Finished Started
```

在这个例子中，主线程和我们创建的非守护线程一退出，守护线程就被中止了。可见，在Groovy中创建线程，我们不需要使用Thread或Runnable的实例。处理线程创建非常简单，而且方便。

### 7.2.3　使用java.io的扩展

java.io包中的File类也加入了很多方法。其中eachFile()和eachDir()（及其变种）这样的方法，可以接受闭包，为目录和文件的导航与迭代提供了方便的方式。

假设我们想读取一个文件的内容。下面是实现该功能的Java代码：

```
// Java代码
import java.io.*;
public class ReadFile {
  public static void main(String[] args) {
    try {
      BufferedReader reader = new BufferedReader(
                      new FileReader("thoreau.txt"));
      String line = null;
      while((line = reader.readLine()) != null) {
        System.out.println(line);
      }
```

```
    } catch(FileNotFoundException ex) {
      ex.printStackTrace();
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

这么读文件可真费劲。Groovy通过向`BufferedReader`、`InputStream`和`File`添加一个`text`属性，使之简单了许多，我们可以把文件的全部内容都读到一个`String`中。这对处理或打印整个输出很有用。下面用Groovy重写了前面的代码：

**ExploringGDK/ReadFile.groovy**

```
println new File('thoreau.txt').text
```

上面代码输出了`thoreau.txt`文件的内容，具体如下：

```
"I went to the woods because I wished to live deliberately,
to front only the essential facts of life, and see if I could
not learn what it had to teach, and not, when I came to die,
to discover that I had not lived..."
- Henry David Thoreau
```

如果不想一次性读入整个文件，而想一次读取并处理一行，可以使用`eachLine()`方法，它会对读入的每行文本调用一个闭包：

**ExploringGDK/ReadFile.groovy**

```
new File('thoreau.txt').eachLine { line ->
  println line // 或者在这里执行自己想对该行进行的任何处理
}
```

如果只是想取得满足某个特定条件的那些行文本，可以使用`filterLine()`，如下所示：

**ExploringGDK/ReadFile.groovy**

```
println new File('thoreau.txt').filterLine { it =~ /life/ }
```

通过前面代码提取出的、过滤后的文本行如下：

```
to front only the essential facts of life, and see if I could
```

我们仅过滤出了输入文件中包含life的行。

如果想使用完毕时自动刷新并关闭一个输入流，可以使用`withStream()`方法。该方法会调用作为参数传入的闭包，并将`InputStream`的实例作为一个参数发送给该闭包。我们一从闭包返回，它就会刷新并关闭这个流。`Writer`有一个类似的方法，叫做`withWriter()`，我们在本节前面部分已经看过一个例子。

InputStream的withReader()会创建一个BufferedReader（被附到输入流上），并将其传给作为参数接受的闭包。也可以通过调用newReader()方法获得一个新的BufferedReader实例。

对于InputStream和DataInputStream中的输入，可以通过调用iterator()方法获得一个Iterator，然后使用该迭代器对输入进行迭代。说到迭代，我们也可以便利地迭代Object InputStream中的对象。

如果想使用Reader代替，也可以。添加到InputStream上的便捷方法，Reader上也有。

在Groovy中可以方便地向文件或流写入内容。OutputStream、ObjectOutputStream和Writer类都通过leftShift()方法（<<操作符）得到了翻新。下面的代码示例使用该操作符向文件中写入信息：

**ExploringGDK/ShiftOperator.groovy**

```
new File("output.txt").withWriter{ file ->
 file << "some data..."
}
```

java.io包中的类还有其他一些扩展，它们使我们的生活更轻松了，编写代码用的时间也更少了。

### 7.2.4　使用java.util的扩展

在第6章中，我们探讨了Groovy对集合类的扩展。这一节，我们来看看java.util包中类的一些其他扩展。

List、Set、SortedMap和SortedSet都加入了asImmutable()方法，用于获得各自实例的一个不可变实例。这些类还加入了一个asSynchronized()方法，用于创建线程安全的实例。

Iterator支持inject()方法，我们在6.4节中讨论过。

java.util.Timer上加入了一个runAfter()方法。使用的语法更为简单，因为该方法接受一个闭包，该闭包将在一个给定的延迟（以毫秒为单位）之后运行。

正如我们在本章所讨论的，Groovy在java.lang.Object层次加入了很多方法。有的方法让我们在调试、日志或获取信息时可以一窥对象内部，有的方法让我们使用一致的接口对待对象和集合，就像组合模式那样。

Object也支持用于元编程的方法，可以动态地访问属性和调用方法。这些方法共同构建起来的高层抽象，减少了日常任务的应用代码的体积，也减少了所需时间。

还可以使用不同类上的专用方法，Groovy为一些类和接口增强了API，比如Matcher、Writer、Reader、List、Map和Socket等，举不胜举。GDK对一些JDK的类和接口也有扩展。

GDK太过庞大，本书无法一一覆盖；可以访问http://groovy.codehaus.org/groovy-jdk，查看全面且持续更新的GDK API列表。

在使用Groovy编程时，我们需要同时参考JDK和GDK。如果在JDK中没有发现要找的东西，一定要记得检查一下GDK，看它是不是支持我们要的特性。

## 7.3 使用扩展模块定制方法

Groovy 2.*x*不只赋予我们使用GDK方法的特权。使用扩展模块（extension-modules）特性，我们还可以在编译时向现有类添加实例方法或静态方法，并在运行时在应用中使用它们。下面通过一个例子，看一下必须遵循的一些简单步骤。

要使用该特性，需要做到两点：首先，想要添加的方法必须定义在一个扩展模块类中；其次，需要在清单文件（manifest）中放一些描述信息，告诉Groovy编译器要查找的扩展模块。

让我们在`String`上创建两个扩展方法，一个是实例方法，一个是静态方法，用于获取给定股票的价格。所有引入的扩展方法，只要基于JDK或GDK，并且将包含这些类的`jar`文件放在它们的`classpath`下，就可以被调用。

两类扩展方法都必须定义为`static`的，而且第一个参数应该是该方法要加入到的类型。定义中还要通过额外的参数来提供该扩展方法要接收的参数。

下面是`String`类上的一个实例扩展方法，写在一个扩展辅助类`PriceExtension`中（这里是用Groovy类编写的，也可以使用其他任何JVM语言编写，包括Java）。

**Extension/com/agiledeveloper/PriceExtension.groovy**

```
package com.agiledeveloper;

class PriceExtension {
  public static double getPrice(String self) {
      def url = "http://ichart.finance.yahoo.com/table.csv?s=$self".toURL()

      def data = url.readLines()[1].split(",")
      Double.parseDouble(data[-1])
    }
}
```

`getPrice()`方法被定义为`static`的，第一个参数说明该方法将被添加到哪个类上。这段代码并没有说出要添加的这个方法是实例方法还是静态方法；这个信息会放在清单声明中，一会我们会看到。

再为同样目的定义一个静态扩展方法。

**Extension/com/agiledeveloper/PriceStaticExtension.groovy**

```
package com.agiledeveloper;

class PriceStaticExtension {
  public static double getPrice(String selfType, String ticker) {
      def url = "http://ichart.finance.yahoo.com/table.csv?s=$ticker".toURL()

      def data = url.readLines()[1].split(",")
      Double.parseDouble(data[-1])
    }
}
```

getPrice()方法接收两个参数，第一个说明该方法要加入到哪个类上；第二个是实际的股票值，说明要获得哪支股票的价格。在第一个版本的getPrice()方法中，股票隐含地包含在实例中；而在这个版本中，股票信息必须作为一个参数传入，因为该方法要运行在String类的静态上下文中。

我们已经准备好了带有扩展方法的辅助类。下面需要声明其存在，并将声明信息和编译好的类打包到一个jar文件中。下面是声明信息，保存在META-INF/services目录下的org.codehaus.groovy.runtime.ExtensionModule文件中：

**Extension/manifest/META-INF/services/org.codehaus.groovy.runtime.ExtensionModule**

```
moduleName=price-module
moduleVersion=1.0-test
extensionClasses=com.agiledeveloper.PriceExtension
staticExtensionClasses=com.agiledeveloper.PriceStaticExtension
```

声明文件中包含4个信息的键值对。moduleName是为模块起的逻辑名称。moduleVersion用于检查该版本是否已经加载。extensionClasses是用逗号分隔的包含实例扩展方法的辅助类的名字。最后，staticExtensionClass是用逗号分隔的包含静态扩展方法的辅助类的名字。

使用下列命令编译这两个辅助类，并创建必要的jar文件：

```
$ groovyc -d classes com/agiledeveloper/*.groovy
$ jar -cf priceExtensions.jar -C classes com -C manifest .
```

priceExtensions.jar文件中包含了编译好的辅助类和清单文件。

再创建一个使用这些扩展方法的示例Groovy文件：

**Extension/FindPrice.groovy**

```
def ticker = "ORCL"

println "Price for $ticker using instance method is ${String.getPrice(ticker)}"
println "Price for $ticker using static method is ${ticker.getPrice()}"
```

我们分别调用了实例方法和静态方法。要加入这些方法，必须将priceExtensions.jar文

件包含在classpath下，像下面命令中这样：

```
$ groovy -classpath priceExtensions.jar FindPrice.groovy
```

Groovy会基于清单文件中提供的信息，无缝地加入扩展方法。下面显示了调用这些扩展方法的结果：

```
Price for ORCL using instance method is 34.75
Price for ORCL using static method is 34.75
```

我们看过了Groovy对JDK方法的扩展，以及如何添加定制的扩展。Groovy还为各种任务提供了强大的类库集。下一章，我们将学习Groovy如何优雅地应对原本乏味的XML处理任务。

第 8 章

# 处理XML

8

处理XML可能很繁琐。使用传统的Java API和库创建和解析XML，往往会让人情绪低落。而使用DOM API在文档层次结构中导航，肯定会让人抓狂。

Groovy缓解了解析和创建XML文档之苦。我们已经看过一些创建XML文档的方式。这一章，我们将再回到这个主题，学习使用3种不同的设施来解析XML文档，它们带来了不同程度的便捷性与效率。我们还将浏览一下Groovy对创建XML文档的支持。

## 8.1 解析 XML

在Groovy中，如果有特殊的需求或原因要依赖较老的API，或者既有代码使用了这些API，那我们可以使用自己熟悉的、基于Java的解析方法与工具。如果已经有了可用的解析XML文档的Java代码，在Groovy中也可以轻松复用。Groovy不会强迫我们重复劳动。

不过，如果要创建新的XML解析代码，Groovy提供的设施则可谓是我们的福音。

在最近的一个项目中，我必须使用来自大约400个XML文档的数据填充一个应用。乍看上去，这个活令人生畏；要处理的文件之多，足以让我望而却步。在快速浏览了一些文件后，我决定使用Groovy处理这些文件、解析XML文档并填充应用。使用`XmlSlurper`类，再结合大约30行Groovy代码，就足以把活干完了。

Groovy解析器相当强大，而且使用方便，它们还支持命名空间，下一节我们将看到。

本章中的所有示例都将会用到下面这个包含一系列语言及其作者信息的XML文档：

**WorkingWithXML/languages.xml**

```
<languages>
  <language name="C++">
    <author>Stroustrup</author>
  </language>
  <language name="Java">
    <author>Gosling</author>
  </language>
```

```
  <language name="Lisp">
    <author>McCarthy</author>
  </language>
  <language name="Modula-2">
    <author>Wirth</author>
  </language>
  <language name="Oberon-2">
    <author>Wirth</author>
  </language>
  <language name="Pascal">
    <author>Wirth</author>
  </language>
</languages>
```

### 8.1.1 使用DOMCategory

使用Groovy的分类（将在13.1节中详细探讨）可以在类上定义动态方法，其中有一个分类可用于处理文档对象模型（Document Object Model，DOM）—— DOMCategory。Groovy还通过添加便捷方法，简化了DOM应用编程接口（API）。

DOMCategory可以通过类GPath（Groovy path expression，即Groovy路径表达式）的符号在DOM结构中导航。

仅通过子元素的名字就能访问所有子元素。例如，不用调用getElementsByTag Name('name')，使用属性name就能获取该元素，就像rootElement.language这样。也就是说，给定根元素rootElement，简单地调用rootElement.language就能获取所有的language元素。[1] DOM解析器会给出rootElement；在下面的例子中，我们将使用DOMBuilder的parse()方法把文档加载到内存中。

在属性名之前放一个@可以获得该属性的值，就像language.@name这样。

在下面的代码中，我们使用DOMCategory在文档中取得语言的名字和作者：

**WorkingWithXML/UsingDOMCategory.groovy**

```
document = groovy.xml.DOMBuilder.parse(new FileReader('languages.xml'))

rootElement = document.documentElement

use(groovy.xml.dom.DOMCategory) {
  println "Languages and authors"
  languages = rootElement.language
```

8

① 这里原文有误，根元素是rootElement，而非languages。从后面的代码也可以知道，languages = root Element.language。——译者注

```
  languages.each { language ->
    println "${language.'@name'} authored by ${language.author[0].text()}"
  }

  def languagesByAuthor = { authorName ->
      languages.findAll { it.author[0].text() == authorName }.collect {
        it.'@name' }.join(', ')
  }

  println "Languages by Wirth:" + languagesByAuthor('Wirth')
}
```

使用这段代码提取出的数据如下：

```
Languages and authors
C++ authored by Stroustrup
Java authored by Gosling
Lisp authored by McCarthy
Modula-2 authored by Wirth
Oberon-2 authored by Wirth
Pascal authored by Wirth
Languages by Wirth:Modula-2, Oberon-2, Pascal
```

`DOMCategory`对于使用DOM API解析XML文档很有用，同时还结合了GPath查询的便捷与Groovy动态特性的优雅。

要使用`DOMCategory`，必须把代码放在`use()`块内。不过本章将会看到的另外两种方法并无这种限制。在前面的示例中，我们使用GPath语法从文档中提取出了想要的细节信息。还写了一个定制的方法（或者说过滤器），用于获得仅由Wirth创造的那些语言。

**GPath是什么？**

与XPath可以帮助导航XML文档的层次结构很类似，GPath可以帮助导航对象（Plain Old Java Object和Plain Old Groovy Object，即POJO和POGO）和XML的层次结构。可以使用句点（.）符号遍历层次结构。例如，`car.engine.power`这种写法会帮助我们通过一个`car`实例的`getEngine()`方法访问其`engine`属性，然后再通过所获得的`engine`实例的`getPower()`方法，访问该实例的`power`属性。如果处理的不是对象，而是XML文档，这种写法会帮助我们获得元素`engine`的一个子元素`power`，而`power`又是元素`car`的一个子元素。与访问元素不同的是，可以使用`car.'@year'`（或`car.@year`）这种写法访问`car`的`year`属性。@符号说明要访问的是属性，而非子元素。

## 8.1.2 使用XMLParser

groovy.util.XMLParser利用了Groovy的动态类型和元编程能力。可以直接使用名字访问文档中的成员。例如，可以使用it.author[0]访问一个作者的名字。

我们使用XMLParser从语言的XML文档中取得想要的数据：

**WorkingWithXML/UsingXMLParser.groovy**

```
languages = new XmlParser().parse('languages.xml')

println "Languages and authors"

languages.each {
  println "${it.@name} authored by ${it.author[0].text()}"
}

def languagesByAuthor = { authorName ->
    languages.findAll { it.author[0].text() == authorName }.collect {
      it.@name }.join(', ')
}


println "Languages by Wirth:" + languagesByAuthor('Wirth')
```

这段代码与我们在“使用DOMCategory”部分看到的示例很像。主要区别在于，这里没有使用use()块。XMLParser 向元素添加了便捷的迭代器，所以可以使用诸如each()、collect()和find()等方法轻松实现导航。

使用XMLParser也有一些不足之处：它没有保留XML InfoSet，而且忽略了文档中的XML注释和处理指令。它带来的便捷性使其成为应对大部分常见处理需求的极好工具。然而，如果有其他特殊需求，我们必须探索更传统的解析器。

## 8.1.3 使用XMLSlurper



对于较大的文档，XMLParser的内存使用可能让人难以忍受。XMLSlurper类可以处理这类情况。它在使用上与XMLParser类似。下面的代码与前面“使用XMLParser”部分的代码几乎相同：

**WorkingWithXML/UsingXMLSlurper.groovy**

```
languages = new XmlSlurper().parse('languages.xml')
println "Languages and authors"

languages.language.each {
  println "${it.@name} authored by ${it.author[0].text()}"
}
```

```
def languagesByAuthor = { authorName ->
    languages.language.findAll { it.author[0].text() == authorName }.collect {
      it.@name }.join(', ')
}
println "Languages by Wirth:" + languagesByAuthor('Wirth')
```

还可以使用XML文档中的命名空间来解析它们。命名空间让我想起一件往事，当时我接到马来西亚一家公司的电话，他们对涉及大量代码以强化测试驱动开发的培训很感兴趣。交谈中我问到，我需要使用什么语言。顿了一下，那位绅士不情愿地说："英语，当然是英语。我们团队里的每个人英语都说得很好。"我的意思实际上是："我要使用什么计算机语言"。这是一个日常交谈中上下文与混淆的例子。XML文档也有同样的问题，而命名空间可以帮助我们处理名字冲突。

记住，命名空间不是URL，但是它们需要保持唯一性。我们在XML文档中使用的命名空间前缀不是独一无二的。我们可以随便起（当然，要有一些命名约束）。因此，要引用查询中的一个命名空间，我们要为其关联一个前缀。可以使用`declareNamespaces()`方法实现这种关联，该方法接受一个以前缀为键，以命名空间为值的映射。一旦定义了前缀，我们的GPath查询也就可以获得名字的前缀了。`element.name`将返回所有带有`name`的子元素，不考虑命名空间；而`element.'ns:name'`则仅返回`ns`关联的命名空间中的元素。来看一个例子，假设我们有一个文档，其中包含计算机语言和自然语言的名字，如下所示：

```
<languages xmlns:computer="Computer" xmlns:natural="Natural">
  <computer:language name="Java"/>
  <computer:language name="Groovy"/>
  <computer:language name="Erlang"/>
  <natural:language name="English"/>
  <natural:language name="German"/>
  <natural:language name="French"/>
</languages>
```

元素名`language`或者是在Computer命名空间中，或者是在Natural命名空间中。下列代码演示了如何同时取得所有语言，以及如何仅取得Natural语言：

**WorkingWithXML/UsingXMLSlurperWithNS.groovy**

```
languages = new XmlSlurper().parse(
  'computerAndNaturalLanguages.xml').declareNamespace(human: 'Natural')

print "Languages: "
println languages.language.collect { it.@name }.join(', ')

print "Natural languages: "
println languages.'human:language'.collect { it.@name }.join(', ')
```

使用这段代码提取出的数据如下：

```
Languages: Java, Groovy, Erlang, English, German, French
Natural languages: English, German, French
```

对于较大的XML文档，我们更愿意使用XMLSlurper。它执行惰性求值，所以内存使用比较友好，而且开销较低。

除了漂亮地解析API，相反的方向，即创建XML文档，Groovy也使其变容易了。下一节我们将看一下不同的创建方式。

## 8.2 创建 XML

在创建业务应用时，我们往往有很多原因要以XML格式呈现数据，比如保存应用状态、与Web服务通信以及表示某些配置文件等。无论需求是什么，Groovy都使创建XML文档变得非常容易了。

在生成XML时，我们可以利用Java API的一切强大之处。如果有特别喜爱的基于Java的XML处理器，比如Xerces，在Groovy中也可有使用。[1]如果我们已经有可用的、以特定格式创建XML文档的Java代码，而且想在我们的Groovy项目中使用它， 这可能是比较好的方法。

如果想使用纯Groovy的方式创建XML文档，可以借助GString在字符串中嵌入表达式的能力，再加上Groovy用于创建多行字符串的设施。对于创建我们可能在代码和测试中需要的小型XML片段，这一设施非常有用。下面是一个简单的例子（更多细节，参见5.3节）：

**WorkingWithStrings/CreateXML.groovy**

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

content = ''
langs.each { language, author ->
  fragment = """
    <language name="${language}">
      <author>${author}</author>
    </language>
  """

  content += fragment
}
xml = "<languages>${content}</languages>"
println xml
```

下面是生成的XML文档：

```
<languages>
  <language name="C++">
    <author>Stroustrup</author>
  </language>
```

8

[1] http://xerces.apache.org/xerces-j

```
  <language name="Java">
    <author>Gosling</author>
  </language>

  <language name="Lisp">
    <author>McCarthy</author>
  </language>
</languages>
```

我们也可以选择`MarkupBuilder`或`StreamingMarkupBuilder`，从任意源创建XML格式的数据输出。这是Groovy应用偏爱的方法， 因为生成器提供的便捷性使创建XML文档变得非常容易。我们不必搞一堆乱七八糟的复杂API或字符串操作，普通、简单的Groovy就够了。我们再来看一个简单的例子（关于使用`MarkupBuilder`和`StreamingMarkupBuilder`的详细信息，参见17.1节）：

**UsingBuilders/BuildUsingStreamingBuilder.groovy**

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

xmlDocument = new groovy.xml.StreamingMarkupBuilder().bind {
  mkp.xmlDeclaration()
  mkp.declareNamespace(computer: "Computer")
  languages {
    comment << "Created using StreamingMarkupBuilder"
    langs.each { key, value ->
      computer.language(name: key) {
        author (value)
      }
    }
  }
}
println xmlDocument
```

这段代码生成的XML文档如下：

```
<?xml version="1.0"?>
<languages xmlns:computer='Computer'>
  <!--Created using StreamingMarkupBuilder-->
    <computer:language name='C++'>
      <author>Stroustrup</author>
    </computer:language>
    <computer:language name='Java'>
      <author>Gosling</author>
    </computer:language>
    <computer:language name='Lisp'>
      <author>McCarthy</author>
    </computer:language>
</languages>
```

如果我们的数据保存在数据库中或Microsoft Excel文件中，可以结合将在第9章中介绍的技术来处理。一旦从数据库中取出数据，就可以应用这里讨论的任何一种方式将其插入到文档中了。

在这一章中，我们知道了Groovy是如何辅助解析XML文档的。Groovy的使用使XML处理变得可以忍受了。如果我们的用户不喜欢维护XML配置文件（谁又喜欢呢），他们可以创建并维护基于Groovy的DSL，再将DSL转为底层框架和库期望的XML格式。如果我们处于接收XML的一端，则可以依赖Groovy为我们提供XML数据的对象表示。

一旦手上有了数据，我们就知道如何使用Groovy将其以XML格式呈现出来。贯穿本书，有很多地方会深入探讨这些主题，后面我们将看到更详细的代码示例。下一章，我们学习如何使用Groovy代码从数据库中获取数据。

# 第 9 章

# 使用数据库

9

我有一个要频繁更新的远程数据库。通过浏览器访问相当慢，不过我已经把更新过程自动化了。我不太情愿用普普通通的Java代码干这个活，因为学不到什么激动人心的东西或新东西。只是在碰到Groovy SQL（GSQL）之前，我得那么做。现在，我的更新是自动化的，而且快速、毫不费力。使用GSQL，更新脚本中的数据要多于代码，信噪比很高。

我们经常会用到数据库，但是很快就会让人感觉乏味无聊。GSQL是JDBC（Java Database Connectivity，Java数据库连接）的包装器，为轻松访问数据提供了很多便捷方法。全部使用Groovy语法，我们可以快速创建SQL查询，然后使用内建的迭代器遍历结果。

这一章我们就来探索GSQL的力量。你将学到编写SQL select查询、从结果生成XML数据、执行数据的插入和更新，还会看到访问Excel文件中数据的方法。

## 9.1 创建数据库

在本章的例子中，我们将使用MySQL；不过我们可以使用任何能够通过JDBC访问的数据库。首先创建将使用于例子中的数据库，同时创建一个名为weather的表。该表中包含的是某些城市的名字和温度值。

使用自动化脚本设置数据库要比手动设置容易。因此，我们创建一个SQL脚本来构建数据库：

```
create database if not exists weatherinfo;
use weatherinfo;

drop table if exists weather;

create table weather (
  city varchar(100) not null,
  temperature integer not null
);

insert into weather (city, temperature) values ('Austin', 48);
insert into weather (city, temperature) values ('Baton Rouge', 57);
insert into weather (city, temperature) values ('Jackson', 50);
```

```
insert into weather (city, temperature) values ('Montgomery', 53);
insert into weather (city, temperature) values ('Phoenix', 67);
insert into weather (city, temperature) values ('Sacramento', 66);
insert into weather (city, temperature) values ('Santa Fe', 27);
insert into weather (city, temperature) values ('Tallahassee', 59);
```

在这个脚本中，我们为一个命名为weather的表定义了模式，并填充了一些示例数据。将该脚本保存到一个名为createdb.sql的文件中，然后使用`mysql--user=root < createdb.sql`命令来运行该脚本，创建数据库。

现在数据库准备好了；接下来，我们看一下从Groovy代码访问数据库的不同方式。

## 9.2　连接到数据库

要连接到数据库，只需要调用`static`的`newInstance()`方法，创建一个`groovy.sql.Sql`类的实例。该方法有个版本接收数据库URL、用户ID、密码和数据库驱动的名字作为参数。如果已经有了一个`java.sql.Connection`实例，或者有一个`java.sql.DataSource`实例，就可以使用`Sql`类的接受相应类型的构造器，而不是使用`newInstance()`。

可以通过调用`Sql`实例的`getConnection()`方法（`connection`属性）获取连接相关信息。在处理结束后，可以通过调用`close()`方法关闭该连接。下面是一个例子，演示了如何连接到我们为本章创建的数据库：

**WorkingWithDatabases/Weather.groovy**

```
def sql = groovy.sql.Sql.newInstance('jdbc:mysql://localhost:3306/weatherinfo',
              userid, password, 'com.mysql.jdbc.Driver')

println sql.connection.catalog
```

该代码报告的数据库名如下：

```
weatherinfo
```

## 9.3　数据库的 Select 操作

我们可以使用`Sql`对象方便地迭代一个表中的数据。只需要调用`eachRow()`方法，为其提供要执行的SQL查询，同时提供用于处理每行数据的闭包，就是这样：

**WorkingWithDatabases/Weather.groovy**

```
println "City                Temperature"
sql.eachRow('SELECT * from weather') {
  printf "%-20s%s\n", it.city, it[1]
}
```

使用前面代码取到的数据如下所示：

```
City                Temperature
Austin              48
Baton Rouge         57
Jackson             50
Montgomery          53
Phoenix             67
Sacramento          66
Santa Fe            27
Tallahassee         59
```

我们让eachRow()在weather表上执行SQL查询，处理该表的所有行。之后对每一行进行迭代（正如名字中的each所示）。还有更Groovy风格的写法，我们可以使用eachRow()提供的GroovyResultSet对象来访问表中的列，可以直接使用列名（如it.city），也可以使用索引（如it[1]）。

在前面的例子中，输出的头部是硬编码的。如果能从数据库中获取，那会更好一些。eachRow()的另一个重载版本会做这么做。它接收两个闭包，一个用于元数据，另一个用于数据。元数据的闭包仅调用一次（在SQL语句执行之后），并以一个ResultSetMetaData实例为参数，而另一个闭包会对结果中的每一行调用一次。我们通过下面的代码尝试一下：

**WorkingWithDatabases/Weather.groovy**

```
processMeta = { metaData ->
  metaData.columnCount.times { i ->
    printf "%-21s", metaData.getColumnLabel(i+1)
  }
  println ""
}

sql.eachRow('SELECT * from weather', processMeta) {
  printf "%-20s %s\n", it.city, it[1]
}
```

输出中显示了使用元数据创建的头部，后面是各行数据：

```
city                 temperature
Austin               48
Baton Rouge          57
Jackson              50
Montgomery           53
Phoenix              67
Sacramento           66
Santa Fe             27
Tallahassee          59
```

如果想处理所有行，但是不想使用迭代器，我们可以在Sql实例上使用rows()方法。该方法返回一个结果数据的ArrayList实例，如下所示：

WorkingWithDatabases/Weather.groovy

```
rows = sql.rows('SELECT * from weather')

println "Weather info available for ${rows.size()} cities"
```

这段代码的输出如下：

```
Weather info available for 8 cities
```

如果改为调用firstRow()方法，则仅得到结果的第一行。我们可以使用Sql的call()方法执行存储过程。使用withStatement()方法，可以设置一个将在查询执行之前调用的闭包。如果想在执行之前拦截并修改SQL查询，该方法会有所帮助。

## 9.4 将数据转为 XML 表示

我们可以从数据库中获得数据，然后使用Groovy生成器创建数据的不同表示。下面这个例子演示了如何创建weather表中数据的一个XML表示（参见17.1节）：

WorkingWithDatabases/Weather.groovy

```
bldr = new groovy.xml.MarkupBuilder()

bldr.weather {
  sql.eachRow('SELECT * from weather') {
    city(name: it.city, temperature: it.temperature)
  }
}
```

这段代码输出的XML如下：

WorkingWithDatabases/Weather.output

```
<weather>
  <city name='Austin' temperature='48' />
  <city name='Baton Rouge' temperature='57' />
  <city name='Jackson' temperature='50' />
  <city name='Montgomery' temperature='53' />
  <city name='Phoenix' temperature='67' />
  <city name='Sacramento' temperature='66' />
  <city name='Santa Fe' temperature='27' />
  <city name='Tallahassee' temperature='59' />
</weather>
```



几乎毫不费力，Groovy和GSQL就帮我们创建了数据库中数据的一个XML表示。

## 9.5　使用 DataSet

在9.3节中，我们看到了如何处理执行一条Select查询所获得的结果集。如果想接收的只是一个过滤后的一些行，比如只要温度值低于32[①]的城市，我们可以相应地设置查询。作为一种选择，我们可以将结果接收为一个groovy.sql.DataSet，以此过滤数据。我们进一步看一下。

Sql类的dataSet()方法接收一个表名，并返回一个虚拟代理——直到迭代时，它才去取实际的行。之后我们可以使用DataSet的each()方法（就像Sql的eachRow()方法）在行上迭代。不过在下面的代码中，我们将使用findAll()方法来过滤结果，只获得温度在零下的城市。当我们调用findAll()时，DataSet会通过基于我们提供的select谓词确定的专用查询做进一步提炼。直到我们在获得的对象上调用each()方法时，才会去取实际的数据。因此，DataSet非常高效，仅提取选中的数据。

**WorkingWithDatabases/Weather.groovy**

```
dataSet = sql.dataSet('weather')
citiesBelowFreezing = dataSet.findAll { it.temperature < 32 }
println "Cities below freezing:"
citiesBelowFreezing.each {
  println it.city
}
```

使用本节介绍的DataSet，这段代码的输出如下：

```
Cities below freezing:
Santa Fe
```

## 9.6　插入与更新

我们可以使用DataSet对象来添加数据，而不仅仅是过滤数据。add()方法接收一个数据的Map，用其中的数据创建一行，如下列代码所示：

**WorkingWithDatabases/Weather.groovy**

```
println "Number of cities : " + sql.rows('SELECT * from weather').size()
dataSet.add(city: 'Denver', temperature: 19)
println "Number of cities : " + sql.rows('SELECT * from weather').size()
```

下面输出说明了这段代码的执行效果：

```
Number of cities : 8
Number of cities : 9
```

① weather表中保存的温度是华氏温度，与摄氏温度的换算关系为：摄氏温度=（华氏温度−32）*5/9。因此华氏温度低于32相当于摄氏温度的零下。——译者注

然而更传统的方式是使用Sql类的execute()或executeInsert()方法，如下所示：

**WorkingWithDatabases/Weather.groovy**

```
temperature = 50
sql.executeInsert("""INSERT INTO weather (city, temperature)
                  VALUES ('Oklahoma City', ${temperature})""")
println sql.firstRow(
  "SELECT temperature from weather WHERE city='Oklahoma City'")
```

前面代码的输出如下：

```
[temperature:50]
```

通过发出相应的SQL命令，也可以以类似方式执行更新和删除操作。

## 9.7 访问 Microsoft Excel

我们还可以使用Sql类来访问Microsoft Excel。如果想了解与COM或ActiveX交互的信息，可以看一下Groovy的Scriptom应用编程接口。[①] 在这一节中，除了用Excel替换掉MySQL，我们将使用已经见过的东西创建一个非常简单的例子。首先，创建一个名为weather.xlsx（如果是老版本的Excel，则是weather.xls）的Excel文件。

把该文件创建在c:\temp目录下。文件中将包含一个名为temperatures的工作表（见工作表底部），数据内容如图9-1所示。访问Excel的代码如下。

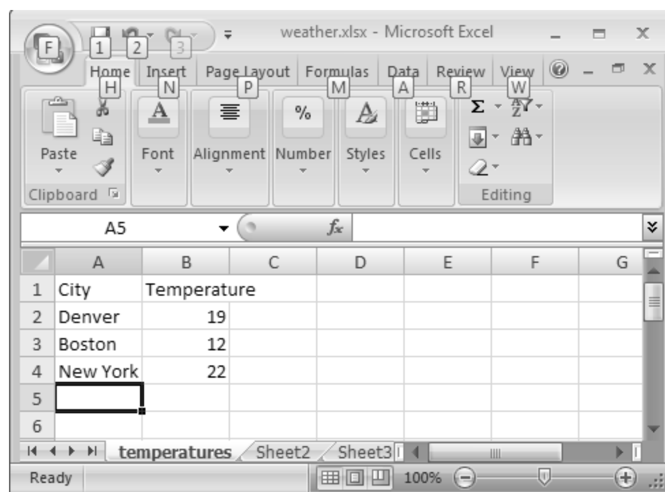



图9-1 我们要使用GSQL访问的一个Excel文件

① http://groovy.codehaus.org/COM+Scripting

**WorkingWithDatabases/Excel/Windows/AccessExcel.groovy**

```
def sql = groovy.sql.Sql.newInstance(
"""jdbc:odbc:Driver=
{Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)};
DBQ=C:/temp/weather.xlsx;READONLY=false""", '', '')

println "City\t\tTemperature"
sql.eachRow('SELECT * FROM [temperatures$]') {
 println "${it.city}\t\t${it.temperature}"
}
```

下面是使用前面的代码从Excel文件中获取到的数据：

```
City            Temperature
Denver          19.0
Boston          12.0
New York        22.0
```

在调用`newInstance()`时，我们指定了Excel的驱动和Excel文件的位置。也可以不这么做，愿意的话，可以为Excel文件设置一个数据源名（Data-Source Name，DSN），并使用古老的JDBC-ODBC驱动桥。

如果那样做的话，我们就不用将文件位置写在代码里了。相反，我们要在Windows上配置数据源名（DSN）。其余执行查询和处理结果的代码都是耳熟能详的了。

在这一章中，我们使用了GSQL来访问关系数据库。对于数据访问，我们可以从该API的简单但强大的功能中受益。只需要几行代码，几分钟的时间，我们的应用就可以读写真正的数据了。

经过这么多章，我们学习了很多API和Groovy编程技术。Groovy的关键优势之一，在于它能够与Java集成和共存。下一章，我们将探讨如何在这两门语言间集成代码。

第 10 章

# 使用脚本和类

10

就算不是最流行的，Java也是最流行的主流企业级语言之一。尽管Groovy可以单独使用，但是我们很可能会混合使用Groovy和Java。在使用Groovy的项目中，Java代码一起演进是这种使用场景很常见。学习如何混合使用以这些语言编写的代码，能够帮助我们在应用中快速采用Groovy。

在Groovy中调用Java代码，非常简单、直接。而乍看上去，在Java中调用Groovy代码就显得没那么简单了。Groovy方法可以接受闭包，Groovy类可以有动态方法，也就是在运行时才出现的方法。Java中是不是可以访问这些东西呢，如果可能的话，又有多困难？我们脑海中蹦出了很多问题。本章将回答这些问题。

我们将看到如何联合编译Java和Groovy代码，如何在Java中使用Groovy代码，以及如何在Java中创建Groovy闭包。我们还将探索如何在Java代码中调用Groovy动态方法，都不用费什么劲。

## 10.1　Java 和 Groovy 的混合

在应用中，我们可以在一个Java类、一个Groovy类或者一个Groovy脚本中实现某个特定功能。之后可以在Java类、Groovy类或Groovy脚本中调用该功能。图10-1展示了混合使用Java类、Groovy类和Groovy脚本的各种选择。

要在Groovy代码中使用Groovy类，无需做什么，直接就可以工作。我们只需要确保所依赖的类在类路径（classpath）下，要么是源代码，要么是字节码。要把一个Groovy脚本拉到我们的Groovy代码中，可以使用`GroovyShell`。而如果要在Java类中使用Groovy脚本，则可以使用JSR 223提供的ScriptEngine API。如果想在Java中使用Groovy类，或者想在Groovy中使用Java类，我们可以利用Groovy的联合编译（joint-compilation）工具。这些都非常简单，本章接下来将一一介绍。

首先，来看一下运行Groovy的各种方法。然后再看一下如何在Java和Groovy中混合使用Groovy的类和脚本。

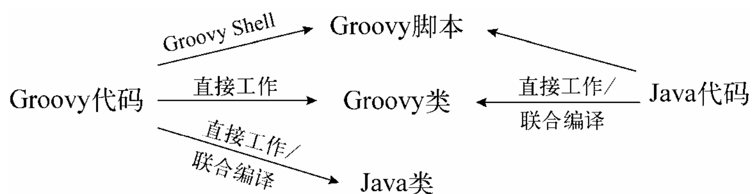



图10-1　混合使用Java类、Groovy类和脚本的方法

## 10.2　运行 Groovy 代码

要运行Groovy代码，我们有两个选择。一个是对源代码运行`groovy`命令。Groovy会自动在内存中编译代码并执行。我们不必引入一个明确的编译步骤。

第二个方法，如果想用类似Java的那种更传统的方法，即显式的编译代码来创建字节码（`.class`文件），可以使用`groovyc`编译器。要执行编译好的字节码，可以像执行编译好的Java代码那样使用`java`命令。唯一的区别是，我们需要把groovy-all-2.1.0.jar文件放在classpath下。记得在classpath里添加一个句点（`.`），这样`java`命令就可以找到当前目录下的类了。这一Java存档文件（JAR）在GROOVY_HOME下的embeddable目录中。作为一个例子，假设我们有一个名为Greet.groovy的文件，其中的代码如下：

**ClassesAndScripts/Greet.groovy**

```
println (['Groovy', 'Rocks!'].join(' '))
```

要运行它，只需要输入`groovy Greet` 。然而，如果想显式地将其编译为Java字节码，则要输入`groovyc Greet.groovy`来创建一个名为Greet.class的文件，文件名正和我们预料的一样。如果代码中有包声明，则该文件会遵照Java的包目录格式在相应的目录下创建。与Groovy类不同的是，Groovy脚本通常没有包声明。可以使用`-d`选项指定非当前目录的目标目录。输入如下命令，可以运行生成的字节码：

```
java -classpath $GROOVY_HOME/embeddable/groovy-all-2.1.0.jar:. Greet
```

在Windows上，请用`%GROOVY_HOME%`替换掉`$GROOVY_HOME`。输出如下：

```
Groovy Rocks!
```

这些步骤说明，可以以字节码形式编译和分发我们的Groovy代码，这与编译和分发Java代码很像。我们可以将其发布为`.class`文件，或者打包成JAR文件。`java`命令看不出区别。如果部署设置有要求的话，可以以这种方法将Groovy代码以字节码形式与项目中的其他字节码一起发布。

下面我们将看一些混合使用Groovy脚本和类的方法。

## 10.3 在 Groovy 中使用 Groovy 类

要在Groovy代码中使用一个Groovy类，只需要确保该类在我们的classpath下。可以使用Groovy源代码，也可以把源代码编译成`.class`文件并使用该文件——随我们选择。当我们的Groovy代码引用了一个Groovy类时，Groovy会以该类的名字在我们的classpath下查找`.groovy`文件；如果找不到，则以同样的名字查找`.class`文件。

假设有一个Groovy源代码文件Car.groovy，内容如下所示，它放在src目录下：

**ClassesAndScripts/src/Car.groovy**

```
class Car
{
  int year = 2008
  int miles

  String toString() { "Car: year: $year, miles: $miles" }
}
```

再假设我们会在一个名为useCar.groovy的文件中使用该类，像下面这样：

**ClassesAndScripts/useCar.groovy**

```
println new Car()
```

要使用这个类，我们输入`groovy -classpath src useCar`。它会自动取到`Car`类的源文件，编译它，创建一个实例，然后生成输出：

```
Car: year: 2008, miles: 0
```

如果我们放的是`Car`的字节码，而不是源代码，步骤是一样的——Groovy可以方便地使用来自`.groovy`或`.class`文件中的类。

如果打算在项目中混合使用Groovy和Java，我们可以借助Groovy提供的联合编译工具，下一节将会看到。

## 10.4 利用联合编译混合使用 Groovy 和 Java

如果Groovy类是预先编译好的，那我们就可以方便地在Java中使用`.class`文件或JAR包。来自Java的字节码和来自Groovy的字节码，对Java而言没什么区别；我们必须把Groovy JAR文件（前面讨论过）放在我们的classpath下，类似于我们使用Spring、Hibernate或其他框架/类库的JAR文件时的做法。

如果我们只有Groovy源代码，而非字节码，又会怎样呢？请记住，当我们的Java类依赖其他Java类时，如果没有找到字节码，`javac`将编译它认为必要的任何Java类。不过`javac`对Groovy

可没这么友好。幸好groovyc支持联合编译。当我们编译Groovy代码时，它会确定是否有任何需要编译的Java类，并负责编译它们。因此我们可以自由地在项目中混合使用Java代码和Groovy代码，而且不必执行单独的编译步骤。简单地调用groovyc就好。

要利用联合编译，我们需要使用-j编译标志。使用-J前缀把标志传给Java编译器。例如，假定我们有一个类，保存在AJavaClass.java文件中：

**ClassesAndScripts/AJavaClass.java**

```
//Java代码
public class AJavaClass {
  {
          System.out.println("Created Java Class");
  }

  public void sayHello() { System.out.println("hello"); }
}
```

我们还有一个保存在UseJavaClass.groovy文件中的脚本，它使用了这个Java类：

**ClassesAndScripts/UseJavaClass.groovy**

```
new AJavaClass().sayHello()
```

要联合编译这两个文件，我们输入这个命令：groovyc -j AJavaClass.java UseJavaClass.groovy -Jsource 1.6。-Jsource 1.6会把可选的选项source = 1.6发送给Java编译器。使用javap检查生成的字节码，会发现AJavaClass作为一个普通的Java类，扩展了java.lang.Object，而UseJavaClass扩展了groovy.lang.Script。

执行这段代码，确认一切正常。尝试下列命令：

```
java -classpath $GROOVY_HOME/embeddable/groovy-all-2.1.0.jar:. UseJavaClass
```

我们应该会看到如下输出：

```
Created Java Class
hello
```

可以与Java无缝地在项目中混用，使Groovy成为企业级应用中可以与Java漂亮集成的一种奇妙语言。我们可以将精力集中在使用每种语言的优点，而不必忙于与任何集成问题做斗争。

集成不仅能在简单情况下简化操作；对于使用了在Java没有直接支持的特性的Groovy代码，我们也可以从Java中调用，下一节我们将看到。

## 10.5　在 Java 中创建与传递 Groovy 闭包

Groovy从诞生的第一天起就支持闭包，但Java还没有认真对待这一理念。令人惊讶的是，得

益于Groovy的动态特性，在Java中创建闭包和调用闭包的Groovy方法非常简单。鉴于Java坚持让我们发送恰当类型的方法实例，Groovy非常友好，而且对于我们使用其特性感到非常开心。

如果仔细检查，我们会发现，当Groovy调用一个闭包时，它只是使用了一个名为`call()`的特殊方法。要在Java中创建一个闭包，我们只需要一个包含该方法的类。如果Groovy代码要向闭包传递实参，我们必须确保`call()`方法接受这些实参作为形参。

在下面的例子中我们将看到，在Java中创建并传递闭包非常简单。我们创建一个Groovy类——`AGroovyClass`，其中有两个接受闭包的方法：

**ClassesAndScripts/AGroovyClass.groovy**

```
class AGroovyClass {
  def useClosure(closure) {
    println "Calling closure"
    closure()
  }

  def passToClosure(int value, closure) {
    println "Simply passing $value to the given closure"
    closure(value)
  }
}
```

`useClosure()`方法会打印一条消息并调用所提供的闭包。`passToClosure()`方法会将它接收到的第一个形参传递给所提供的闭包。

要在Java中调用`useClosure()`方法，我们需要提供一个实现了`call()`方法的实例，像这样：

**ClassesAndScripts/UseAGroovyClass.java**

```
//Java代码
public class UseAGroovyClass {
  public static void main(String[] args) {
    AGroovyClass instance = new AGroovyClass();
    Object result = instance.useClosure(new Object() {
      public String call() {
        return "You called from Groovy!";
      }
    });

    System.out.println("Received: " + result);
  }
}
```

Java和Groovy代码既可以联合编译，也可以分别编译。要联合编译，我们使用`groovyc -j UseAGroovyClass.java AGroovy- Class.groovy`命令。然后我们可以使用`java -classpath $GROOVY_HOME/embeddable/groovy-all-2.1.0.jar:. UseAGroovyClass`命令运行Java代码。

我们在Java中创建的匿名类的实例会被无缝地传递给Groovy，反过来，调用也会回到该匿名类：

```
Calling closure
Received: You called from Groovy!
```

调用一个接受形参的闭包也没有很大不同，我们会在passToClosure()方法中的调用中看到：

**ClassesAndScripts/UseAGroovyClass2.java**

```
//Java代码
System.out.println("Received: " +
  instance.passToClosure(2, new Object() {
    public String call(int value) {
      return "You called from Groovy with value " + value;
    }
  }));
```

Java中这一版的call()方法接收一个形参，passToClosure()方法会在Groovy段给该参数赋一个值，我们在输出中会看到：

```
Simply passing 2 to the given closure
Received: You called from Groovy with value 2
```

我们必须确保call()方法接受恰当数目和类型的形参。剩下的具体细节，Groovy会为我们处理。

本节探讨了在Java中调用Groovy闭包。反方向的调用也是如此简单。在http://www.jroller.com/melix/entry/coding_a_groovy_closure_in中，Cédric Champeau演示了如何把一个Java方法当作Groovy端的一个闭包。

我们看过了如何调用带闭包的方法；下面来看一下如何从Java中调用Groovy动态方法。

## 10.6　在 Java 中调用 Groovy 动态方法

Groovy可以在运行时创建方法（第三部分将会介绍）。这些方法不能从Java中直接调用，因为在编译时，这些代码在字节码中还不存在。它们在运行时产生，但是如果我们要从Java中调用它们，需要在编译时编写调用语句（或者使用反射）。要调用动态方法，我们必须先通过Java编译器的编译，然后运行时才能进行分派。这听上去很复杂，但是要相信Groovy！

每个Groovy对象都实现了GroovyObject接口，该接口有一个叫作invokeMethod()的特殊方法。该方法接受要调用的方法的名字，以及要传递的参数。在Java这一端，invokeMethod()可以用来调用Groovy中使用元编程动态定义的方法。

我们通过例子来看一下。创建一个Groovy类，其中包含一个特殊方法——method Missing()，当某个不存在的方法被调用时，该方法会介入。

ClassesAndScripts/DynamicGroovyClass.groovy

```
class DynamicGroovyClass {
  def methodMissing(String name, args) {
    println "You called $name with ${args.join(', ')}."
    args.size()
  }
}
```

这个类完全是动态的，除了methodMissing()，它没有实际的方法。因为这个类可以接受任何方法调用，所以我们几乎可以在它上面调用任何方法。要从Java端调用我们期望的方法，可调用invokeMethod()，并将方法的名字和一个由参数组成的数组传给它，如下面的示例所示。

ClassesAndScripts/CallDynamicMethod.java

```
public class CallDynamicMethod {
  public static void main(String[] args) {
    groovy.lang.GroovyObject instance = new DynamicGroovyClass();

    Object result1 = instance.invokeMethod("squeak", new Object[] {});
    System.out.println("Received: " + result1);

    Object result2 =
      instance.invokeMethod("quack", new Object[] {"like", "a", "duck"});
    System.out.println("Received: " + result2);
  }
}
```

我们创建了一个DynamicGroovyClass实例，并将其赋值给了一个GroovyObject类型（所有Groovy对象都支持）的引用。使用这个引用，我们可以在该类上调用任何方法，包括动态的和预定义的。一旦Groovy接收到这些方法，它会通过常规的Groovy方法分派过程来处理方法调用，11.1节会介绍此过程。Groovy会响应我们从Java端进行的调用，从下面的输出可以看出。invokeMethod()会将被调用方法返回的任何响应信息返回到Java端。

```
You called squeak with .
Received: 0
You called quack with like, a, duck.
Received: 3
```

如果Groovy因为某些原因无法执行被调用的方法，或者该方法没有执行成功，调用invokeMethod()将会失败。请准备好处理该方法可能会抛出的异常。

从Java中可以使用任何Groovy类，这点没有限制，不管它们是否是动态的。下面我们来看一下从Java中使用Groovy类的情况。

## 10.7　在 Groovy 中使用 Java 类

在Groovy中使用Java类简单且直接。如果我们想使用的Java类是JDK的一部分，可以像在Java中那样导入这些类或它们的包。Groovy默认会导入很多包和类（参见2.1节），因此，如果我们想使用的类已经导入（比如`java.util.Date`），直接用就可以了，无需再导入。

如果想使用一个自己的Java类，或者不是标准JDK中的类，在Groovy中可以像在Java中那样导入它们。请确保导入了必要的包或类，或者使用类的全限定名来引用它们。当运行`groovy`时，可以使用`-classpath`选项指定`.class`文件或JAR包的路径。如果类文件和我们的Groovy代码在同一目录下，则不需要通过该选项指定目录。

我们看一个例子。假定我们有一个名为`GreetJava`的Java类，它从属于`com.agiledeveloper`包，而且有一个`static`方法`sayHello()`，如下所示：

**ClassesAndScripts/GreetJava.java**

```
// Java代码
package com.agiledeveloper;

public class GreetJava {
        public static void sayHello() {
                System.out.println("Hello Java");
        }
}
```

我们想从一个Groovy脚本中调用该方法，因此首先编译Java类`GreetJava`，这样会在./com/agiledeveloper目录下生成类文件GreetJava.class，其中的句点（.）是当前目录。然后在UseGreetJava.groovy文件中创建一个Groovy脚本，内容如下：

**ClassesAndScripts/UseGreetJava.groovy**

```
com.agiledeveloper.GreetJava.sayHello()
```

要运行该脚本，只需要输入`groovy UseGreetJava`。该脚本会顺利运行，并调用`GreetJava`类中的`sayHello()`方法，如下列输出所示：

```
Hello Java
```

如果类文件不在当前目录下，我们仍然可以使用它，只是需要记得设置`-classpath`选项。假定类文件GreetJava.class位于~/release/com/agiledeveloper目录下，其中~是我们的home目录。

要运行前面提到的Groovy脚本（`UseGreetJava.groovy`），请使用下面的命令：

```
$groovy -classpath ~/release UseGreetJava
```

在这个例子中，我们显式地编译了Java代码，然后在Groovy脚本中使用了字节码。如果想显

式地编译Groovy代码，不必分别编译Java和Groovy。可以使用联合编译代替。

并非所有的Groovy代码都需要显式编译。Groovy脚本一般通过groovy命令使用。下面我们将介绍如何混合使用Groovy脚本。

## 10.8 从 Groovy 中使用 Groovy 脚本

Groovy脚本保存的语句和表达式不必局限于源代码中某个特定类。我们可以直接使用groovy命令执行这些脚本。通过GroovyShell类，我们还可以从其他的Groovy脚本和类中调用它们。下面看一个例子：

**ClassesAndScripts/Script1.groovy**

```
println "Hello from Script1"
```

这里有一个命名为Script1.groovy的文件，我们想将其作为另一个脚本——Script2.groovy——的一部分来执行，如下所示：

**ClassesAndScripts/Script2.groovy**

```
println "In Script2"
shell = new GroovyShell()
shell.evaluate(new File('Script1.groovy'))

// 或是更简单点
evaluate(new File('Script1.groovy'))
```

输出如下：

```
In Script2
Hello from Script1
Hello from Script1
```

使用GroovyShell，我们可以在任何文件（或字符串）中对脚本调用evaluate()方法，以执行该脚本。这很容易。但是（凡事总有例外），如果我们想向脚本传递一些参数，又该怎么办呢？

**ClassesAndScripts/Script1a.groovy**

```
println "Hello ${name}"
name = "Dan"
```

这个脚本需要一个变量name。我们可以使用一个Binding实例来绑定变量，如下所示：

**ClassesAndScripts/Script2a.groovy**

```
println "In Script2"

name = "Venkat"

shell = new GroovyShell(binding)
```

```
result = shell.evaluate(new File('Script1a.groovy'))

println "Script1a returned : $result"
println "Hello $name"
```

在发起调用的脚本中，我们创建了一个变量name（与被调用脚本中用到的变量名字相同）。当创建GroovyShell的实例时，我们将当前的Binding对象传给了它（每个脚本执行时都有一个这样的对象）。被调用脚本现在就可以使用（读取和设置）发起调用脚本所知道的变量了。

前面代码的输出如下：

```
In Script2
Hello Venkat
Script1a returned : Dan
Hello Dan
```

如果脚本会返回一个值，我们还可以通过evaluate()方法的返回值得到它，如前面例子所示。

在前面的例子中，我们将发起调用的脚本的Binding对象传给了GroovyShell。如果不希望影响当前的binding， 而是想将其与被调用脚本的binding分开，我们只需要创建一个新的Binding实例，在该实例上调用setProperty()来设置变量名和值，并将其作为创建GroovyShell实例时的一个参数，如下所示：

**ClassesAndScripts/Script3.groovy**

```
println "In Script3"

binding1 = new Binding()
binding1.setProperty('name', 'Venkat')
shell = new GroovyShell(binding1)
shell.evaluate(new File('Script1a.groovy'))

binding2 = new Binding()
binding2.setProperty('name', 'Dan')
shell.binding = binding2
shell.evaluate(new File('Script1a.groovy'))
```

这段代码的输出如下：

```
In Script3
Hello Venkat
Hello Dan
```

如果想向脚本传递一些命令行参数，可以使用GroovyShell类的run()方法来代替evaluate()方法。

利用GroovyShell，可以轻松加载任何脚本，并将其作为我们的Groovy代码的一部分来执行。这一特性非常有用，有些例行任务可能会保存在可复用的脚本中，该特性可以帮我们运行这些任

务。此外，该特性还可以帮我们构建并执行DSL。

我们知道了如何从Groovy代码中调用Groovy脚本，下面来看一下如何从Java代码内调用Groovy脚本。

## 10.9 从 Java 中使用 Groovy 脚本

如果想在Java中使用Groovy脚本，我们可以利用JSR 223（JSR即Java Specification Request，Java规范请求）。

JSR 223在JVM和脚本语言（参见附录A中的Java脚本程序员指南）之间架起了一座桥梁。它为Java和一些实现了JSR 223脚本引擎API的脚本语言提供了一种标准的交互方式。在Java 5中我们可以下载并使用JSR 223，而Java 6已经默认包含了JSR 223。

与Groovy相比，其实JSR 223这种选择更适合JVM上的其他语言，Groovy对Java和Groovy的联合编译功能降低了对类似工具的需求。

要从Java中调用一个（非编译好的）脚本，需要使用脚本引擎。而脚本引擎可以通过调用`ScriptEngineManager`的`getEngineByName()`方法获得。从Java代码中执行脚本，则要调用脚本引擎的`eval()`方法。要使用Groovy脚本，需要确保.../jsr223-engines/groovy/build/groovy-engine.jar在我们的classpath下。

我们来看一个例子：从Java中执行一个小的Groovy脚本。（在Java中，需要处理我们并不关心的异常，这种“乐趣”随之而来。本章其余的例子不会演示异常处理代码，不过你要记得在需要的地方加上。）

**MixingJavaAndGroovy/CallingScript.java**

```
// Java代码
package com.agiledeveloper;
import javax.script.*;

public class CallingScript {
  public static void main(String[] args) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine engine = manager.getEngineByName("groovy");
    System.out.println("Calling script from Java");
    try {
      engine.eval("println 'Hello from Groovy'");
    } catch(ScriptException ex) {
      System.out.println(ex);
    }
  }
}
```

输出如下：

```
Calling script from Java
Hello from Groovy
```

在这个例子中，Groovy脚本被嵌在了传给`eval()`方法的字符串参数中。现实中的脚本一般和这个例子不同，不会采用硬编码方式。脚本可能在文件中，在输入流中，在对话框中，等等。如果是那样的情况，我们会使用`eval()`方法的其他可以使用`Reader`的重载版本。

如果脚本会向发起调用的Java程序返回任何结果，我们可以使用`eval()`方法的`Object`返回值来接收。

使用这种方法，我们可以在Java应用内调用任何Groovy脚本。如果想向脚本传递一些参数，比如说从Java中创建但要在Groovy中访问的一个Java对象，我们可以使用`Bindings`。

`Bindings`是一个`Map<String, Object>`，它使得对象可以通过一个具名的值获得。`ScriptContext`允许脚本引擎连接到宿主应用中的Java对象，比如`Bindings`。我们可以显式地访问这些对象并与其交互，也可以简单地在`ScriptEngine`实例上使用`get()`和`put()`。如果想执行同样的脚本，但想为变量提供一组不同的值，我们可以创建不同的上下文，并在调用`eval()`时使用这些上下文。

下面看一个从Java中向Groovy脚本传递参数的例子：

**MixingJavaAndGroovy/ParameterPassing.java**

```
engine.put("name", "Venkat");
engine.eval("println \"Hello ${name} from Groovy\"; name += '!' ");
String name = (String) engine.get("name");
System.out.println("Back in Java:" + name);
```

这段代码的输出如下：

```
Hello Venkat from Groovy
Back in Java:Venkat!
```

我们使用`put()`方法向脚本引擎发送了一个`String`对象（其值为`Venkat`）。同时将名字`name`用于绑定变量。脚本内使用了`name`这个变量。我们还可以修改它的值。在Java端，通过在脚本引擎上调用`get()`方法，可以获得该变量的当前值。

JSR 223提供了调用实例方法的能力，还支持调用没有与任何特定的类关联的函数。可以分别使用`Invocable`的`invokeMethod()`和`invokeFunction()`这两个方法。如果打算重复使用一个脚本，可以使用`Compilable`接口，以避免重复编译该脚本。

不同于使用`ScriptEngineManager`，从Java内还可以使用`GroovyScriptEngine`，这和从Groovy内使用`GroovyShell`很像。[1]`GroovyScriptEngine`的`run()`方法接受一个脚本文件名和一

① http://groovy.codehaus.org/Embedding+Groovy

个`binding`变量，其中该变量负责映射脚本需要的参数。如果脚本改变了，它甚至可以重新加载并重新运行脚本，这点使它非常适合嵌入在Java服务器应用中。

我们通常会将Java代码编译成`.class`文件并打成JAR包。要使用其他的Java类，只需要将其`.class`文件，或者包含这些类的JAR包放在我们的classpath下。如果从Groovy中调用Java类，Groovy差不多也是这样要求。Groovy的联合编译也让我们过得更轻松了。利用这一设施，我们可以并排地使用Groovy和Java代码，此外还可以在同一个对象上使用这两种语言无缝地进行调试和处理。

本章探讨的是，我们可以非常容易地引入和使用Groovy脚本。贯穿本书，我们会看到很多使用来自JDK的Java类的例子。本章还解决了如何在应用中使用自己的Java类和Groovy类的问题。混合使用Java和Groovy来创建企业级应用完全没有障碍。

谈到企业级应用，下一章我们将看到，动态、灵活的Groovy语言在元编程领域也大有作为。

# Part 3 第三部分

# MOP 与元编程

本部分内容

# 第 11 章 探索元对象协议

在Java中，使用反射可以在运行时探索程序的结构，以及程序的类、类的方法、方法接受的参数。然而，我们仍然局限于所创建的静态结构。我们无法在运行时修改一个对象的类型，或是让它动态获得行为——至少现在还不能。想象一下，如果可以基于应用的当前状态，或基于应用所接受的输入，动态地添加方法和行为，代码会变得更灵活，我们的创造力和开发效率也会提高。好了，不用再想象了——在Groovy中，元编程就提供了这一功能。

使用这些特性设计的应用，可扩展性会有多好呢？非常好。我最近有机会给一家公司做咨询，该公司在创建Web应用时，正在从Java转向使用Groovy和Grails。他们的产品部署后，需要在线上进行某些定制。在现有的系统中，这要耗去一些开发人员和测试人员数周的时间和精力。通过与其核心开发者紧密合作，我们使用Groovy元编程和一些后台服务设法实现了自动化定制，而且立竿见影，应用的吞吐量和员工的开发效率都得到了提高。

*元编程*（metaprogramming）意味着编写能够操作程序的程序，包括操作程序自身。像Groovy这样的动态语言通过元对象协议（MetaObject Protocol，MOP）提供了这种能力。利用Groovy的MOP，创建类、编写单元测试和引入模拟对象都很容易。

在Groovy中，使用MOP可以动态调用方法，甚至在运行时合成类和方法。该特性让我们有这种感觉：对象顺利地修改了它的类。比如Grails/GORM就使用了该特性，为数据查询合成方法。借助MOP，在Groovy中可以创建内部的领域特定语言（参见第19章）。Groovy生成器（参见第17章）也依赖MOP。因此，MOP是我们要学习和探索的最重要的概念之一。本章和后面几章将研究MOP中的一些概念。

本章将通过两个方面来探索MOP，一个是Groovy对象的组成，一个是Groovy如何解析Java对象和Groovy对象的方法调用。然后我们会看一下查询方法和属性的不同方式，最后探讨如何动态访问对象。

一旦消化吸收了本章中的这些基础，你就为学习第12章中的如何拦截方法调用做好了准备。第13章和第14章将探索如何在运行时向类中注入和合成方法。最后，我们将以第15章结束MOP相关的讨论。

## 11.1 Groovy 对象



Groovy提供的灵活性一开始可能会让人困惑， 因此如果想充分利用MOP，我们需要理解Groovy对象和Groovy的方法处理。

Groovy对象是带有附加功能的Java对象。在Groovy中，Groovy对象比编译好的Java对象具有更多的动态行为。此外，对于Java对象和Groovy对象上的方法调用，Groovy的处理方式也是不同的。

在一个Groovy应用中，我们会使用三类对象：POJO、POGO和Groovy拦截器。POJO（Plain Old Java Object）就是普通的Java对象，可以使用Java或JVM上的其他语言来创建。POGO（Plain Old Groovy Object）是用Groovy编写的对象，扩展了`java.lang.Object`，同时也实现了`groovy.lang.GroovyObject`接口。Groovy拦截器是扩展了`GroovyInterceptable`的Groovy对象，具有方法拦截功能，一会我们会讨论。Groovy这样定义的`GroovyObject`接口：

```
//这是Groovy源代码中GroovyObject.java的一个片段
package groovy.lang;
public interface GroovyObject {
    Object invokeMethod(String name, Object args);
    Object getProperty(String property);
    void setProperty(String property, Object newValue);
    MetaClass getMetaClass();
    void setMetaClass(MetaClass metaClass);
}
```

`invokeMethod()`、`getProperty()`和`setProperty()`使Java对象具有了高度的动态性。可以使用它们来处理运行中创建的方法和属性。`getMetaClass()`和`setMetaClass()`使创建代理拦截POGO上的方法调用、在POGO上注入方法变得非常容易。一旦一个类被加载到JVM中，我们就不能修改它的元对象`Class`了。不过我们可以通过调用`setMetaClass()`修改它的`MetaClass`。这给我们一种对象在运行时修改了它的类的感觉。

下面我们看一下`GroovyInterceptable`接口。它是一个扩展了`GroovyObject`的标记接口，对于实现了该接口的对象而言，其上的所有方法调用，不管是存在的还是不存在的，都会被它的`invokeMethod()`方法拦截。

```
//这是Groovy源代码中GroovyInterceptable.java的一个片段

package groovy.lang;

public interface GroovyInterceptable extends GroovyObject {
}
```

Groovy支持对POJO和POGO进行元编程。对于POJO，Groovy维护了`MetaClass`的一个`MetaClassRegistry`，如图11-1所示。另一方面，POGO有一个到其`MetaClass`的直接引用。

当我们调用一个方法时，Groovy会检查目标对象是一个POJO还是一个POGO。对于不同的对象类型，Groovy的方法处理是不同的。

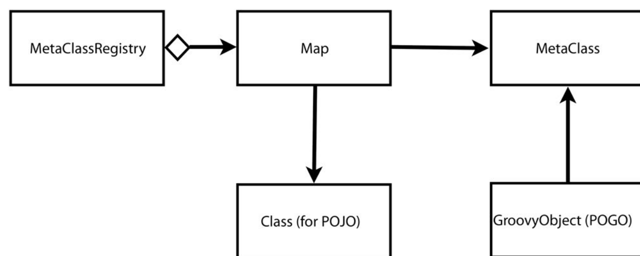



图11-1　POJO、POGO及其MetaClass

对于一个POJO，Groovy会去应用类（application-wide）的`MetaClassRegistry`取它的`MetaClass`，并将方法调用委托给它。因此，我们在它的`MetaClass`上定义的任何拦截器或方法，都优先于POJO原来的方法。

对于一个POGO，Groovy会采取一些额外的步骤，如图11-2所示。如果对象实现了`GroovyInterceptable`，那么所有的调用都会被路由给它的`invokeMethod()`。在这个拦截器内，我们可以把调用路由给实际的方法，使类AOP（Aspect-Oriented Programming，面向方面编程）的操作成为可能。

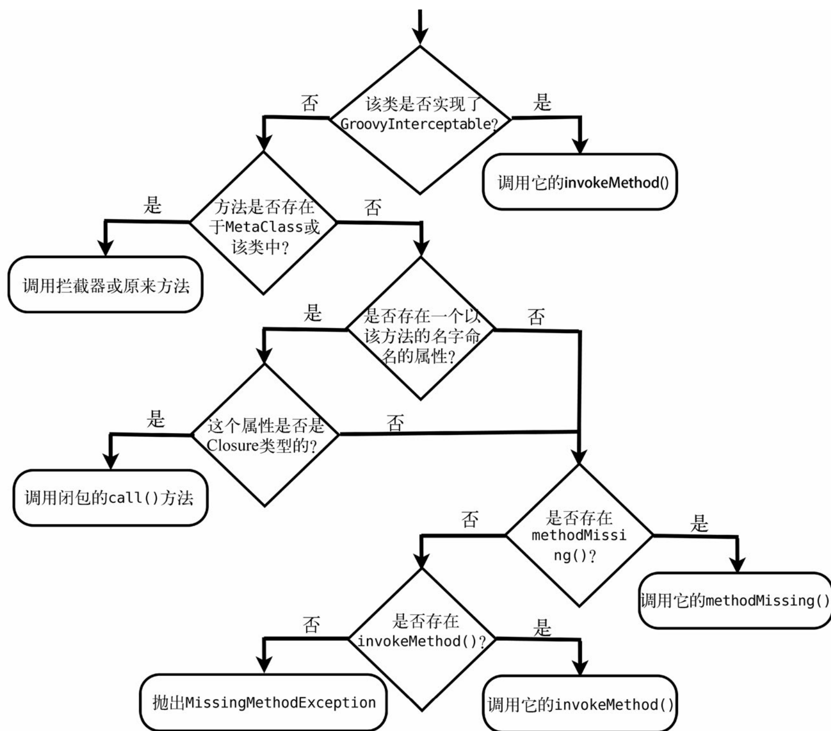



图11-2　Groovy如何处理POGO上的方法调用

如果该POGO没有实现`GroovyInterceptable`，那么Groovy会先查找它的`MetaClass`中的方法，之后，如果没有找到，则查找POGO自身上的方法。如果该POGO没有这样的方法，Groovy会以方法名查找属性或字段。如果属性或字段是`Closure`类型的，Groovy会调用它，替代方法调用。如果Groovy没有找到这样的属性或字段，它会做最后两次尝试。如果POGO有一个名为`methodMissing()`的方法，则调用该方法。否则调用POGO的`invokeMethod()`。如果我们在POGO上实现了这个方法，它会被调用。`invokeMethod()`的默认实现会抛出一个`MissingMethod Exception`异常，说明调用失败。

现在，在代码中看一下前面讨论的机制，使用带有不同选项的类来说明Groovy的方法处理。请研究代码，并尝试确定每种情况下Groovy会执行哪个方法（在通读代码的同时，请参考图11-2）：

**ExploringMOP/TestMethodInvocation.groovy**

```
class TestMethodInvocation extends GroovyTestCase {
  void testInterceptedMethodCallonPOJO() {
    def val = new Integer(3)
    Integer.metaClass.toString = {-> 'intercepted' }

    assertEquals "intercepted", val.toString()
  }

  void testInterceptableCalled() {
    def obj = new AnInterceptable()
    assertEquals 'intercepted', obj.existingMethod()
    assertEquals 'intercepted', obj.nonExistingMethod()
  }

  void testInterceptedExistingMethodCalled() {
    AGroovyObject.metaClass.existingMethod2 = {-> 'intercepted' }
    def obj = new AGroovyObject()
    assertEquals 'intercepted', obj.existingMethod2()
  }

  void testUnInterceptedExistingMethodCalled() {
    def obj = new AGroovyObject()
    assertEquals 'existingMethod', obj.existingMethod()
  }

  void testPropertyThatIsClosureCalled() {
    def obj = new AGroovyObject()
    assertEquals 'closure called', obj.closureProp()
  }

  void testMethodMissingCalledOnlyForNonExistent() {
    def obj = new ClassWithInvokeAndMissingMethod()
    assertEquals 'existingMethod', obj.existingMethod()
    assertEquals 'missing called', obj.nonExistingMethod()
```

```
  }

  void testInvokeMethodCalledForOnlyNonExistent() {
    def obj = new ClassWithInvokeOnly()
    assertEquals 'existingMethod', obj.existingMethod()
    assertEquals 'invoke called', obj.nonExistingMethod()
  }

  void testMethodFailsOnNonExistent() {
    def obj = new TestMethodInvocation()
    shouldFail (MissingMethodException) { obj.nonExistingMethod() }
  }
}

class AnInterceptable implements GroovyInterceptable {
  def existingMethod() {}
  def invokeMethod(String name, args) { 'intercepted' }
}

class AGroovyObject {
  def existingMethod() { 'existingMethod' }
  def existingMethod2() { 'existingMethod2' }
  def closureProp = { 'closure called' }
}

class ClassWithInvokeAndMissingMethod {
  def existingMethod() { 'existingMethod' }
  def invokeMethod(String name, args) { 'invoke called' }
  def methodMissing(String name, args) { 'missing called' }
}

class ClassWithInvokeOnly {
  def existingMethod() { 'existingMethod' }
  def invokeMethod(String name, args) { 'invoke called' }
}
```

下列输出证实，所有测试通过，而且Groovy正是像我们讨论的那样处理方法：

```
.........
Time: 0.047

OK (9 tests)
```

## 11.2　查询方法与属性

在运行时，可以查询一个对象的方法和属性，以确定该对象是否支持某一特定行为。对于要在运行时动态添加的行为，这一点尤为有用。不仅可以向类添加行为，还可以向类的一些实例添加行为。

可以使用MetaObjectProtocol的getMetaMethod()（MetaClass扩展了MetaObjectProtocol）来获得一个元方法（metamethod）。如果要找的是一个static方法，可以使用getStaticMetaMethod()。要获得重载方法的列表，则使用这些方法的复数形式——getMetaMethods()和getStaticMetaMethods()。类似地，使用getMetaProperty()和getStaticMetaProperty()可以获得一个元属性（metaproperty）。如果只是想简单地检查是否存在，而非获得元方法和元属性，我们使用respondsTo()检查方法，使用hasProperty()检查属性。

根据Groovy文档，MetaMethod"表示一个Java对象上的方法，除了不使用反射调用该方法，有点像Method[1]"。如果有一个方法名字符串，我们可以调用getMetaMethod()，并使用获得的MetaMethod去调用方法，像这样：

**ExploringMOP/UsingMetaMethod.groovy**

```
str = "hello"
methodName = 'toUpperCase'
// 名字可能来自输入，而不是硬编码的

methodOfInterest = str.metaClass.getMetaMethod(methodName)

println methodOfInterest.invoke(str)
```

动态调用的方法产生如下输出：

```
HELLO
```

我们不用在编写代码时就知道方法的名字。可以通过输入获得名字并动态调用该方法。

要确定对象是否响应方法调用，可以使用respondsTo()方法。它接收的参数包括我们要查询的实例、要查询的方法的名字，以及以逗号分隔的提供给该方法的参数。它会返回一个MetaMethod列表，其中包括匹配的方法。我们通过一个例子来使用一下：

**ExploringMOP/UsingMetaMethod.groovy**

```
print "Does String respond to toUpperCase()? "
println String.metaClass.respondsTo(str, 'toUpperCase')? 'yes' : 'no'

print "Does String respond to compareTo(String)? "
println String.metaClass.respondsTo(str, 'compareTo', "test")? 'yes' : 'no'

print "Does String respond to toUpperCase(int)? "
println String.metaClass.respondsTo(str, 'toUpperCase', 5)? 'yes' : 'no'
```

下面是这段代码的输出：

---

①这里的Method指的是java.lang.reflect.Method。——译者注

```
Does String respond to toUpperCase()? yes
Does String respond to compareTo(String)? yes
Does String respond to toUpperCase(int)? no
```

getMetaMethod()和respondsTo()很方便。对于要查找的某个方法，我们可以简单地把该方法的参数发送给这些方法。getMetaMethod()和respondsTo()不会像Java反射中的getMethod()方法那样坚持让我们传入参数的Class的数组。更棒的是，如果我们感兴趣的方法不接收任何形参，那就不用发送任何实参，连null都不需要。这是因为，这些方法的最后一个形参是一个形参数组，而Groovy将其看作可选的。

在前面的代码中还发生了一件神奇的事情：我们使用了Groovy对boolean的特殊处理（更多信息可参见2.7节）。respondsTo()方法返回的是MetaMethod的一个列表，因为我们在条件语句中使用了该结果（?:操作符），如果其中存在任何方法，Groovy会返回true，否则返回false。因此，我们不必显式地检查所返回列表的大小是否大于零——Groovy会帮我们做这件事。

## 11.3　动态访问对象

除了前面介绍的方法和属性的查询方式和动态调用方式，在Groovy中，还有其他比较方便的访问属性和调用方法的方式。现在我们就以一个String实例为例，看一下这些方式。假设我们在运行时通过输入获得属性和方法的名字，然后想动态访问它们。下面是一些方式。

**ExploringMOP/AccessingObject.groovy**

```groovy
def printInfo(obj) {
  // 假定用户从标准输入键入这些值
  usrRequestedProperty = 'bytes'
  usrRequestedMethod = 'toUpperCase'

  println obj[usrRequestedProperty]
  //或
  println obj."$usrRequestedProperty"

  println obj."$usrRequestedMethod"()
  //或
  println obj.invokeMethod(usrRequestedMethod, null)
}

printInfo('hello')
```

下面是这段代码的输出：

```
[104, 101, 108, 108, 111]
[104, 101, 108, 108, 111]
HELLO
HELLO
```

要动态调用一个属性，我们可以使用索引操作符[]，或使用点记号后跟一个计算属性名的GString，就像前面代码中那样。要调用一个方法，在对象上使用点记号或调用invokeMethod，并将方法名和实参列表传给它（这种情况下是null）。

要迭代一个对象的所有属性，我们可以使用properties属性（或getProperties()方法），如下所示：

**ExploringMOP/AccessingObject.groovy**

```
println "Properties of 'hello' are: "
'hello'.properties.each { println it }
```

输出如下：

```
Properties of 'hello' are:
class=class java.lang.String
bytes=[B@74f2ff9b
empty=false
```

本章介绍了Groovy元编程的一些基础知识。有了这一基础，下面几章将进一步探索MOP，理解Groovy如何工作，以及如何充分利用MOP思想。

第 12 章

# 使用MOP拦截方法

12

在Groovy中可以相当轻松地实现面向方面编程（Aspect-Oriented Programming，即AOP），比如方法拦截或方法建议[①]。有三种类型的建议。当然不是我们每天收到的好建议、坏建议和不请自来的建议。我们将关注的是前置建议（Before Advice）、后置建议（After Advice）和环绕建议（Around Advice）。前置建议是我们想在某个特定操作之前执行的代码，后置建议在某个操作执行之后才执行，而环绕建议则用于代替指定的操作执行。在Groovy中，使用MOP就可以实现这些建议类型或拦截器，而不需要任何复杂的工具或框架。

这里我们将讨论Groovy中拦截方法调用的两种方式：要么让对象拦截，要么让MetaClass拦截。让对象处理的话，需要实现GroovyInterceptable接口。当我们不是类的作者，或者该类是一个Java类，再或者我们想动态地引入拦截，则第二种方式更适合。本章将介绍这两种方式。还有一种拦截方法的方式——使用分类（Categories），将在13.1节中再讨论这一点。

## 12.1 使用 GroovyInterceptable 拦截方法

如果一个Groovy对象实现了GroovyInterceptable接口，那么当调用该对象上的任何一个方法时——不管是存在的还是不存在的，被调用的都会是invokeMethod()。也就是说，GroovyInterceptable的invokeMethod()方法劫持了该对象上的所有方法调用。

如果想实现环绕建议，只需要在这个方法内实现我们的逻辑。然而想实现前置建议或后置建议（或者想都实现），我们首先要实现自己的前置/后置逻辑，然后在恰当的时机将调用路由到实际方法。要路由调用，我们将使用MetaMethod，它可以从MetaClass获得（参见11.2节）。

**invokeMethod、GroovyInterceptable和GroovyObject**

如果一个Groovy对象实现了GroovyInterceptable接口，在调用它的任何方法时，都会调用到其invokeMethod()方法。对于其他的Groovy对象，只有调用到不存在的方法时才

① 关于AOP的深入讨论，可以参见Ramnivas Laddad的*AspectJ in Action* [Lad03]一书。

会调用该方法。有个例外，如果我们在一个对象的MetaClass上实现了invokeMethod()，不管方法存在与否，都会调用到该方法。



假设我们想在调用某个类的一些方法之前运行过滤器，比如验证、登录确认和日志等。我们不希望手工编辑每一个要调用过滤器的方法，因为这种工作是重复的，冗长乏味，而且容易出错。我们也不希望让方法的调用者去调用过滤器，因为无法保证他们是否调用。拦截过滤器的方法调用是个不错的选择。这将是无缝的，而且是自动化的。

在本章的例子中，我们用System.out.println()替换了println()，以避免用于提供信息的打印消息被拦截。这是因为println()是Groovy在Object上注入的一个方法，如果调用它，我们编写的代码会将其拦截，而System.out.println()是PrintStream上的一个static方法，不会受到影响。

下面看一个例子：在Car类中，我们想在其他任何方法执行之前运行一个过滤器方法check()。下面是使用GroovyInterceptable实现该功能的代码：

**InterceptingMethodsUsingMOP/InterceptingCalls.groovy**

```
Line 1 class Car implements GroovyInterceptable {
     -   def check() { System.out.println "check called..." }
     -
     -   def start() { System.out.println "start called..." }
     5
     -   def drive() { System.out.println "drive called..." }
     -
     -   def invokeMethod(String name, args) {
     -     System.out.print("Call to $name intercepted... ")
    10
     -     if (name != 'check') {
     -       System.out.print("running filter... ")
     -       Car.metaClass.getMetaMethod('check').invoke(this, null)
     -     }
    15
     -     def validMethod = Car.metaClass.getMetaMethod(name, args)
     -     if (validMethod != null) {
     -       validMethod.invoke(this, args)
     -     } else {
    20       Car.metaClass.invokeMethod(this, name, args)
     -     }
     -   }
     - }
     -
    25 car = new Car()
     -
```

```
- car.start()
- car.drive()
- car.check()
30 try {
-   car.speed()
- } catch(Exception ex) {
-   println ex
- }
```

下列输出说明方法被正确拦截：

```
Call to start intercepted... running filter... check called...
start called...
Call to drive intercepted... running filter... check called...
drive called...
Call to check intercepted... check called...
Call to speed intercepted... running filter... check called...
groovy.lang.MissingMethodException:
  No signature of method: Car.speed()
  is applicable for argument types: () values: []
```

因为Car类实现了GroovyInterceptable接口，Car实例上的所有方法调用都会被其invokeMethod()方法拦截。在那个方法中，如果方法名不是check，我们就调用前置过滤器，也就是check()方法。借助MetaClass的getMetaMethod()来确定被调用方法是不是一个存在的合法方法，如果是，就使用MetaMethod的invoke()调用这个方法，如代码的第18行所示。

如果没找到这个方法，就简单地将调用请求路由给MetaClass，如第20行所示。这为动态合成的方法创造了机会，我们将在14.1节介绍。如果该方法仍然不存在，MetaClass的invokeMethod()将抛出一个MissingMethodException异常。

这个例子中创建了一个前置建议。将指定代码放到第18行之后，可以轻松实现后置建议。如果想实现环绕建议，可以删除第18行的代码，或者将其替换为候补代码。

## 12.2　使用 MetaClass 拦截方法

上一节使用GroovyInterceptable拦截了方法调用，这种方式适用于拦截作者是我们自己的类中的方法。然而，如果我们无权修改类的源代码，或者这个类是个Java类，就行不通了。此外，我们也有可能在运行时决定基于某些条件或应用状态开始拦截调用。对于这几种情况，我们可以在MetaClass上实现invokeMethod()方法，并以此来拦截方法。

现在我们使用MetaClass重写上一节的例子。在这个版本中，Car没有实现GroovyInterceptable接口，也没有提供invokeMethod()方法。而且即便它提供了这个方法，但是Car没有实现GroovyInterceptable接口，添加到MetaClass中的invokeMethod()方法也会优先被调用。代码如下：

**InterceptingMethodsUsingMOP/InterceptingCallsUsingMetaClass.groovy**

```
Line 1  class Car {
     -    def check() { System.out.println "check called..." }
     -
     -    def start() { System.out.println "start called..." }
     5
     -    def drive() { System.out.println "drive called..." }
     -  }
     -
     -  Car.metaClass.invokeMethod = { String name, args ->
    10    System.out.print("Call to $name intercepted... ")
     -
     -    if (name != 'check') {
     -      System.out.print("running filter... ")
     -      Car.metaClass.getMetaMethod('check').invoke(delegate, null)
    15    }
     -
     -    def validMethod = Car.metaClass.getMetaMethod(name, args)
     -    if (validMethod != null) {
     -      validMethod.invoke(delegate, args)
    20    } else {
     -      Car.metaClass.invokeMissingMethod(delegate, name, args)
     -    }
     -  }
     -
    25
     -  car = new Car()
     -
     -  car.start()
     -  car.drive()
    30  car.check()
     -  try {
     -    car.speed()
     -  } catch(Exception ex) {
     -    println ex
    35  }
```

观察一下输出中的方法拦截：

```
Call to start intercepted... running filter... check called...
start called...
Call to drive intercepted... running filter... check called...
drive called...
Call to check intercepted... check called...
Call to speed intercepted... running filter... check called...
groovy.lang.MissingMethodException:
  No signature of method: Car.speed()
  is applicable for argument types: () values: []
```

12

在第9行，我们以闭包的形式实现了invokeMethod()，并将其设置到Car的MetaClass上。现在这个方法会拦截Car实例上的所有调用。这个版本的invokeMethod()和上一节我们在Car上实现的版本有两点不同。第一点，这里使用的是delegate，而非this（比如第14行）。在用于拦截的闭包内，delegate指向的是要拦截其方法的目标对象。第二点，在第21行，我们调用的是MetaClass上的invokeMissingMethod()方法，而非invokeMethod()。因为这里已经在invokeMethod()方法中，所以不应该递归地调用该方法了。

我们前面看过，使用MetaClass拦截调用有一个好处，那就是也可以拦截POJO上的调用。为了看看实际效果，我们拦截一下Integer上的方法调用，并执行类AOP的建议：

**InterceptingMethodsUsingMOP/InterceptInteger.groovy**

```
Integer.metaClass.invokeMethod = { String name, args ->
  System.out.println("Call to $name intercepted on $delegate... ")

  def validMethod = Integer.metaClass.getMetaMethod(name, args)
  if (validMethod == null) {
    Integer.metaClass.invokeMissingMethod(delegate, name, args)
  } else {
    System.out.println("running pre-filter... ")
    result = validMethod.invoke(delegate, args) // 如果要实现环绕建议，则去掉这条语句

    System.out.println("running post-filter... ")
    result
  }
}
println 5.floatValue()
println 5.intValue()
try {
  println 5.empty()
} catch(Exception ex) {
  println ex
}
```

输出说明了Integer上的方法拦截：

```
Call to floatValue intercepted on 5...
running pre-filter...
running post-filter...
5.0
Call to intValue intercepted on 5...
running pre-filter...
running post-filter...
5
Call to empty intercepted on 5...
groovy.lang.MissingMethodException:
  No signature of method: java.lang.Integer.empty()
  is applicable for argument types: () values: []
```

我们在Integer的MetaClass上添加的invokeMethod()，拦截了5这个Integer实例上的方法调用。要拦截任何Object上的方法调用，而不是仅限于Integer，应该将拦截器添加到Object的MetaClass上。

如果感兴趣的只是拦截对不存在的方法的调用，那应该使用methodMissing()来代替invokeMethod()。这一点我们将在第14章学习。

我们可以在MetaClass上同时提供invokeMethod()和methodMissing()。invokeMethod()会优先于methodMissing()处理。然而，在调用invokeMissingMethod()时，我们其实就是在调用methodMissing()来处理不存在的方法。[①]

能够使用MetaClass拦截方法调用是受到了Grails的影响。该特性最初是在Grails中引入的，后来被移到了Groovy中[②]。下面花一分钟时间检查一下 MetaClass这个带给我们巨大力量的类：

**InterceptingMethodsUsingMOP/ExamineMetaClass.groovy**

```
Integer.metaClass.invokeMethod = { String name, args -> /* */ }
println Integer.metaClass.getClass().name
```

输出如下：

```
groovy.lang.ExpandoMetaClass
```

ExpandoMetaClass是MetaClass接口的一个实现，也是Groovy中负责实现动态行为的关键类之一。通过向该类添加方法可以实现向类中注入行为，甚至可以使用该类特化单个对象。

这里有一个与ExpandoMetaClass有关的陷阱。该类是MetaClass接口的众多不同实现之一。默认情况下，Groovy目前并没有使用ExpandoMetaClass。当我们向metaClass中添加一个方法时，默认的metaClass会被用一个ExpandoMetaClass实例替换掉。

下面的例子演示了这一行为。在动态添加方法之前，我们检查一下实例的metaclass。

**InterceptingMethodsUsingMOP/MetaClassUsed.groovy**

```
def printMetaClassInfo(instance) {
  print "MetaClass of ${instance} is ${instance.metaClass.class.simpleName}"
  println " with delegate ${instance.metaClass.delegate.class.simpleName}"
}

printMetaClassInfo(2)
println "MetaClass of Integer is ${Integer.metaClass.class.simpleName}"
println "Adding a method to Integer metaClass"
Integer.metaClass.someNewMethod = { -> /* */ }
```

---

① invokeMissingMethod()方法的作用就是尝试调用methodMissing()，失败则抛出异常。具体请参考Groovy的文档。——译者注

② http://graemerocher.blogspot.com/2007/06/dynamic-groovy-groovys-equivalent-to.html

```
printMetaClassInfo(2)
println "MetaClass of Integer is ${Integer.metaClass.class.simpleName}"

@groovy.transform.Immutable
class MyClass {
  String name
}

obj1 = new MyClass("obj1")

printMetaClassInfo(obj1)
println "Adding a method to MyClass metaClass"
MyClass.metaClass.someNewMethod = { -> /* */}
printMetaClassInfo(obj1)

println "obj2 created later"
obj2 = new MyClass("obj2")
printMetaClassInfo(obj2)
```

从输出可以看到，Groovy切换了开始时的默认`metaClass`。

```
MetaClass of 2 is HandleMetaClass with delegate MetaClassImpl
MetaClass of Integer is HandleMetaClass
Adding a method to Integer metaClass
MetaClass of 2 is HandleMetaClass with delegate ExpandoMetaClass
MetaClass of Integer is ExpandoMetaClass
MetaClass of MyClass(obj1) is HandleMetaClass with delegate MetaClassImpl
Adding a method to MyClass metaClass
MetaClass of MyClass(obj1) is HandleMetaClass with delegate MetaClassImpl
obj2 created later
MetaClass of MyClass(obj2) is HandleMetaClass with delegate ExpandoMetaClass
```

开始时，`Integer`实例所用的`metaClass`是一个`HandleMetaClass`实例，还有一个用于委托方法调用的隐藏的`MetaClassImpl`实例。当我们向`Integer`的`metaClass`添加方法时，它被一个`ExpandoMetaClass`实例替换掉了。完成添加之后再次查询，我们看到，`Integer`实例的`metaClass` 的`delegate` 变成了一个`ExpandoMetaClass`实例，不再是原来的`MetaClassImpl`。对于我们自己的Groovy类，the MetaClass used for instances created before the query for metaClass on our class is different from the instances created after we added a method. 在使用元编程时，这种行为会引发一些令人奇怪的现象。我们会在13.2节和14.1节看到相关例子。如果Groovy一致地使用`ExpandoMetaClass`作为默认实现就好了。是否应该这样修改，Groovy社区也有相关讨论。

本章介绍了如何拦截方法调用，并以此来实现类AOP的方法建议能力。该特性还有其他一些用武之地，比如为测试创建模拟方法，临时替换掉有问题的方法，在不修改现有代码的条件下研究算法的替代实现，等等。我们可以通过动态添加方法进一步了解MOP。下一章将探索这一主题。

第 13 章

# MOP方法注入

Java程序员常有这样的抱怨："要是String类支持一个encrypt()方法，那就方便了。"面向对象编程就是针对可扩展性的，但是具体能够扩展到何种程度，往往要受到语言的限制。如果可以打开任何类，根据需要加入想要的方法，会怎么样呢？那样无疑会带来无限的可扩展性，编写有表现力的代码也将非常轻松。在Groovy中就可以这么做，而且毫不费力。

在Groovy中可以随时打开一个类。也就是说，可以动态地向类中添加方法，允许它们在运行时改变行为。这样就不再是使用一个静态结构和一组预先定义好的方法，对象变得敏捷且灵活，而且可以基于应用中的现实情况引入行为。比如，可以根据所接收到的某个特定输入添加一个方法。能够修改类的行为，是元编程和Groovy元对象协议（MOP）的核心。

Groovy的MOP支持以下3种技术注入行为中的任何一种：

- 分类（category）
- ExpandoMetaClass
- Mixin

本章将探讨使用这些技术进行方法注入的MOP设施。

## 13.1 使用分类注入方法

Groovy的分类提供了一种可控的方法注入方式——方法注入的作用可以限定在一个代码块内。分类（category）是一种能够修改类的MetaClass的对象，而且这种修改仅在代码块的作用域和执行线程内有效，当退出代码块时，一切会恢复原状。分类可以嵌套，也可以在一个代码块内应用多个分类。本节将通过一些例子探索分类的行为与使用。

假设有一个保存在String或StringBuilder中的社会保险号。现在想注入一个toSSN()方法，来负责以xxx-xx-xxxx格式返回字符串。下面讨论几种实现方式。

比如说第一个方案是创建一个扩展了StringBuilder的类——SSNStringBuilder，它提供了toSSN()方法。遗憾的是，StringBuilder的使用者可享受不到这个方法，因为该方法在

SSNStringBuilder上才有。此外，我们也无法扩展*final*的String类，进而也无法加入该方法。

但是，我们可以利用Groovy的分类：创建一个StringUtil类，并加入一个静态的toSSN()方法。该方法接受一个参数，即要将该方法注入的目标对象。它会查验字符串的大小，并以指定格式返回一个字符串。要使用这个新方法，又需要调用一个特殊方法——use()，它接受两个参数：一个分类，一个闭包代码块，要注入的方法就在该代码块内生效。

代码如下：

**InjectionAndSynthesisWithMOP/UsingCategories.groovy**

```
class StringUtil {
  def static toSSN(self) { //如果想将参数限制为String类型，则使用toSSN(String self)
    if (self.size() == 9) {
      "${self[0..2]}-${self[3..4]}-${self[5..8]}"
    }
  }
}
use(StringUtil) {
  println "123456789".toSSN()
  println new StringBuilder("987654321").toSSN()
}

try {
  println "123456789".toSSN()
} catch(MissingMethodException ex) {
  println ex.message
}
```

执行注入的方法，查看输出：

```
123-45-6789
987-65-4321
No signature of method: java.lang.String.toSSN()
  is applicable for argument types: () values: []
Possible solutions: toSet(), toSet(), toURI(),
  toURL(), toURL(), toURI()
```

这里注入的方法仅在use块内有效，当在其外调用toSSN()时，会出现MissingMethodException异常。

在块内调用String和StringBuilder实例上的toSSN()，会被路由到分类StringUtil中的静态方法。toSSN()的self参数会被指派为目标实例。因为我们没有定义self参数的类型，其类型默认为Object，toSSN()可以在任何对象上使用。如果想限制该方法仅支持String和StringBuilder，则必须使用显式的参数类型创建两个版本的toSSN()，一个是String self，一个是StringBuilder self。

如果使用前面例子中的语法，Groovy的分类要求注入的方法是静态的，而且至少接受一个参

数。第一个参数（这个例子中叫作self）指向的是方法调用的目标。要注入的方法所需的参数都放在后面。参数可以是任何合法的Groovy参数，包括对象和闭包。

Groovy还为分类提供了一种可选的语法。与其亲自编写静态方法，还不如让Groovy编译器使用前面讨论的格式将实例方法转变为静态方法。此举可以使用一个特殊的@Category注解来实现，比如实现StringUtil可以像下面这样：

**InjectionAndSynthesisWithMOP/UsingCategories.groovy**

```
@Category(String)
class StringUtilAnnotated {
  def toSSN() {
    if (size() == 9) {
      "${this[0..2]}-${this[3..4]}-${this[5..8]}"
    }
  }
}
use(StringUtilAnnotated) {
  println "123456789".toSSN()
}
```

@Category注解会根据我们传入的String参数将新定义的StringUtilAnnotated类的toSSN()转变为public static toSSN(String self) {...}。这个分类的使用方式和前面一样，代码的输出如下：

```
123-45-6789
```

注解式语法减少了一些程序化的东西，不必再将分类的方法声明为静态的，也不必再额外传递第一个参数。然而，如果使用这种注解语法，方法就被限制为只能使用参数中指定的类型（这个例子中是String），对于其他类型——比如StringUtil——是不可复用的，除非我们指定更为一般化的参数，如Object。

现在花点时间来理解一下前面的例子，当调用use()时，背后到底有何魔法。Groovy将在脚本中调用的use()方法路由到了GroovyCategorySupport类的public static Object use(Class categoryClass, Closure closure)方法。该方法定义了一个新的作用域，其中包括栈上的一个新的属性/方法列表，用于目标对象的MetaClass。之后它会检查给定分类中的每个静态方法，并将静态方法及其参数（至少有一个）加入到属性/方法列表中。最后，它会调用附在其后的闭包。在该闭包内的任何方法调用都会被拦截，然后发送给由分类提供的实现（如果存在的话）。对于我们加入的新方法，以及拦截的已有方法，均是如此。最后，一旦等到从闭包返回，use()就结束掉前面创建的作用域，丢弃分类中注入的方法。

注入的方法可以以对象或闭包为参数，下面这一例子说明这一点。再来编写另一个分类——FindUtil，它提供了一个extractOnly()方法，用以提取由闭包参数指定的部分字符串：

**InjectionAndSynthesisWithMOP/UsingCategories.groovy**

```
class FindUtil {
  def static extractOnly(String self, closure) {
    def result = ''
    self.each {
      if (closure(it)) { result += it }
    }
    result
  }
}
use(FindUtil) {
  println "121254123".extractOnly { it == '4' || it == '5' }
}
```

前面调用的结果如下：

```
54
```

**内置分类**

Groovy提供了很多方便我们使用的分类。DOMCategory（参见8.1.1节）可以帮助我们像JavaBean那样处理DOM对象，还支持使用GPath（参见8.1.1节）。ServletCategory使用JavaBean惯例提供了Servlet API对象的属性。

可以同时使用多个分类，带入多组方法。use()可以接受一个分类，也可以接受一个由分类组成的列表。下面的例子同时使用了前面创建的两个分类：

**InjectionAndSynthesisWithMOP/UsingCategories.groovy**

```
use(StringUtil, FindUtil) {
  str = "123487651"
  println str.toSSN()
  println str.extractOnly { it == '8' || it == '1' }
}
```

输出如下：

```
123-48-7651
181
```

尽管use()可以接受由Class实例组成的一个List，但是Groovy也非常愿意接受以逗号分隔的类名列表。这是因为，类一旦定义，Groovy会将类名转变为一个对该类的Class元对象的引用。例如，String等价于String.class；换句话说，String == String.class。

当我们混合使用多个分类时，有一个最明显的问题：在存在方法名冲突时，以什么样的顺序确定调用的是哪个方法。答案是：列表中的最后一个分类优先级最高。

use()可以嵌套调用。也就是说，可以在一个use()调用所使用的闭包内调用另一个use()。内部的分类优先于外部的。

以上就是如何向一个已有的类注入新方法。第12章介绍的拦截已有方法的不同方式中，也提到可以使用分类拦截方法。假设我们想拦截对toString()的调用，然后在响应内容的每一侧加上两个感叹号。下面是使用分类的实现方式：

InjectionAndSynthesisWithMOP/UsingCategories.groovy

```
class Helper {
  def static toString(String self) {
    def method = self.metaClass.methods.find { it.name == 'toString' }
    '!!' + method.invoke(self, null) + '!!'
  }
}

use(Helper) {
  println 'hello'.toString()
}
```

这段代码的输出如下：

```
!!hello!!
```

Helper的toString()用于拦截对字符串hello上相应方法的调用。然而，在这个拦截器内，我们又想调用原来的toString()，这里使用String类的MetaClass获得了它。

使用分类拦截方法调用，不像第12章中的其他方式那样优雅，不能使用分类来过滤一个实例上的所有方法调用，所以要为想拦截的每个方法编写单独的方法。此外，当存在嵌套的分类时，因为内部的分类的优先级高，对于同样的方法，无法利用顶层的分类进行拦截。因此，应该将分类用于方法注入，而不是方法拦截。

分类提供了一个漂亮的方法注入协议。其效果包含在use()块内的控制流中。一旦离开代码块，注入的方法就消失了。当在方法上接受了一个参数时，我们可以对这个参数应用自己的分类。那感觉就像扩充了所接受对象的类型。而当我们离开方法时，这个对象的类也不会受到影响。利用不同的分类，可以实现不同版本的拦截或注入的方法。

然而分类还有一些限制。其作用限定在use()块内，所以也就限定于执行线程。因此注入的方法也是有限制的。已有的方法可以在任何地方调用，但注入的方法只能在use()块内调用。多次进入和退出这个块是有代价的。每次进入时，Groovy都必须检查静态方法，并将其加入到新作用域的一个方法列表中。在块的最后还要清理该作用域。

如果调用不是太频繁，而且想要分类这种可控的方法注入所提供的隔离性，就可以使用分类。如果这些特性变成了限制，则可以使用ExpandoMetaClass来注入方法。下一节我们将讨论。

## 13.2　使用 ExpandoMetaClass 注入方法

要创建领域特定语言（DSL），需要能够向不同的类、甚至类的层次结构中加入任意的方法。为了保持一定的流畅性，我们需要注入实例方法和静态方法，操纵构造器，以及将实例方法转变为属性。在创建代替协作者的模拟对象时，也需要这些能力。本节将讨论如何修改和增强类的结构。

通过向类的MetaClass添加方法可以实现向类中注入方法。不同于分类中的局限于一个块，这种注入方法是全局可用的。利用12.2节探讨的ExpandoMetaClass可以添加方法、属性、构造器和静态方法，也可以从其他类借方法，甚至向POGO和POJO中注入方法。

下面这个例子使用ExpandoMetaClass向Integer中注入一个名为EdaysFromNow()的方法。我们希望5.daysFromNow()返回从今天开始计算5天之后的日期。代码如下：

**InjectionAndSynthesisWithMOP/UsingExpandoMetaClass.groovy**

```
Integer.metaClass.daysFromNow = { ->
  Calendar today = Calendar.instance
  today.add(Calendar.DAY_OF_MONTH, delegate)
  today.time
}
println 5.daysFromNow()
```

输出如下：

```
Thu Sep 20 13:16:03 MST 2012
```

这段代码使用一个闭包实现了daysFromNow()，然后将其引入到了Integer的MetaClass中。（要将该方法注入到任何对象中，可以将其添加到Object的MetaClass中。）在这个闭包内，需要获得Integer的目标对象。delegate引用的就是该目标。关于委托和闭包的讨论，参见4.9节和7.1节。

去掉方法调用尾部的括号，会看上去更流畅（参见19.2节），接着就可以调用5.daysFromNow了。然而这需要一个小窍门（参见19.9节）。如果没有括号，Groovy会将方法当成一个属性，所以需要设置一个属性，而非方法。要定义一个名为daysFromNow的属性，必须创建一个名为getDaysFromNow()的方法，像下面这样：

**InjectionAndSynthesisWithMOP/UsingExpandoMetaClass.groovy**

```
Integer.metaClass.getDaysFromNow = { ->
  Calendar today = Calendar.instance
  today.add(Calendar.DAY_OF_MONTH, delegate)
  today.time
}
println 5.daysFromNow
```

前面代码的输出如下。对daysFromNow属性的调用现在被路由到了getDaysFromNow()方法。

```
Thu Sep 20 13:16:03 MST 2012
```

Integer上已经注入了一个方法，但是与其类似的Short和Long这些类上并没有这个方法。该怎么做呢？当然谁也不想重复地把这个方法加到那些类上。一种思路是把这个闭包保存到一个变量中，然后将其指派给那些类，如下所示：

**InjectionAndSynthesisWithMOP/MethodOnHierarchy.groovy**

```
daysFromNow = { ->
  Calendar today = Calendar.instance
  today.add(Calendar.DAY_OF_MONTH, (int)delegate)
  today.time
}

Integer.metaClass.daysFromNow = daysFromNow
Long.metaClass.daysFromNow = daysFromNow

println 5.daysFromNow()
println 5L.daysFromNow()
```

输出如下：

```
Thu Sep 20 13:26:43 MST 2012
Thu Sep 20 13:26:43 MST 2012
```

也可以选择在Integer的基类Number上提供这个方法。我们在Number上添加一个名为someMethod()的方法，看看在Integer和Long上面是不是可以调用：

**InjectionAndSynthesisWithMOP/MethodOnHierarchy.groovy**

```
Number.metaClass.someMethod = { ->
  println "someMethod called"
}

2.someMethod()
2L.someMethod()
```

如下所示，上述代码的输出证实，派生类上可以调用注入的方法：

```
someMethod called
someMethod called
```

知道了如何向类层次结构中注入一个方法后，可能也想把方法引入到一个接口层次结构中，这样所有实现该接口的类就都可以使用这个方法了。19.11节将看到如何向接口中添加方法。

向类中注入静态方法也是可以的，只需要将其加入到MetaClass的static属性中。

下面向Integer中加入一个静态方法：

**InjectionAndSynthesisWithMOP/UsingExpandoMetaClass.groovy**

```
Integer.metaClass.'static'.isEven = { val -> val % 2 == 0 }
```

```
println "Is 2 even? " + Integer.isEven(2)
println "Is 3 even? " + Integer.isEven(3)
```

执行这段代码，将得到如下输出：

```
Is 2 even? true
Is 3 even? false
```

解决了注入实例方法和静态方法的问题，类中还可以有第三类方法，即构造器。通过定义一个名为constructor的特殊属性可以加入构造器。因为我们是要添加一个构造器，而不是替换一个现有的，所以使用了<<操作符。注意：使用<<来覆盖现有的构造器或方法，Groovy会报错。下面例子为Integer引入一个构造器，它接受一个Calendar，这样该实例就可以保存从这一年的开始到指定日期的天数了。

**InjectionAndSynthesisWithMOP/UsingExpandoMetaClass.groovy**

```
Integer.metaClass.constructor << { Calendar calendar ->
   new Integer(calendar.get(Calendar.DAY_OF_YEAR))
}

println new Integer(Calendar.instance)
```

前面代码的输出如下：

```
349
```

在注入的构造器中，使用了Integer中现有的一个接受int的构造器。可以直接返回调用Calendar的get()的结果，而不是创建一个新的Integer实例。在这种情况下，自动装箱会负责创建一个Integer实例。请务必确认该实现没有递归调用自身，否则会导致StackOverflowError。

如果不是想添加一个新的构造器，而是想替换（或者说覆盖，尽管严格讲构造器是不可覆盖的）一个原来的，可以使用=操作符代替<<操作符。

**InjectionAndSynthesisWithMOP/UsingExpandoMetaClass.groovy**

```
Integer.metaClass.constructor = { int val ->
  println "Intercepting constructor call"
  constructor = Integer.class.getConstructor(Integer.TYPE)
  constructor.newInstance(val)
}

println new Integer(4)
println new Integer(Calendar.instance)
```

这段代码的输出如下：

```
Intercepting constructor call
4
Intercepting constructor call
349
```

在覆盖的构造器内仍然可以使用反射调用原来的实现。可以看到，其他构造器——不管是预先定义的还是注入的——仍是完好无损的。因此，当我们使用一个Calendar实例创建Integer时，它使用了之前注入的构造器，而这个构造器现在反过来又使用了这里提供的覆盖的构造器。

如果想添加一两个方法，使用ClassName.metaClass.method = {...}这样的语法向metaClass中添加，既简单又方便。如果想添加一堆方法，这样声明和设置很快就会感觉费劲了。Groovy提供了一种可以将这些方法分组的方式，组织成一种叫作ExpandoMetaClass（EMC）DSL的方便的语法。前面的例子逐一地向Integer的metaClass中添加了一些方法。作为替代，可以将这些方法分组，如下所示：



**InjectionAndSynthesisWithMOP/UsingEMCDSL.groovy**

```
Integer.metaClass {
    daysFromNow = { ->
      Calendar today = Calendar.instance
      today.add(Calendar.DAY_OF_MONTH, delegate)
      today.time
    }

    getDaysFromNow = { ->
      Calendar today = Calendar.instance
      today.add(Calendar.DAY_OF_MONTH, delegate)
      today.time
    }

    'static' {
      isEven = { val -> val % 2 == 0 }
    }

    constructor = { Calendar calendar ->
      new Integer(calendar.get(Calendar.DAY_OF_YEAR))
    }

    constructor = { int val ->
      println "Intercepting constructor call"
      constructor = Integer.class.getConstructor(Integer.TYPE)
      constructor.newInstance(val)
    }
}
```

在一个传给ClassName.metaClass的闭包中，对想注入类的metaClass中的方法进行分组。将每个实例方法的代码包在一个闭包中，然后将其赋值给想注入的方法名。要注入静态方法，则定义一个以单词static为前缀的闭包，并将静态方法的定义放在这个闭包内，如例子中所示。要定义一个构造器，和前面一样，使用constructor。

EMC DSL减少了代码噪音，它将加入到一个类中的一堆方法集中在一起，因此也更容易看清楚。

ExpandoMetaClass在注入方法方面非常灵活。可以从应用中的任何地方调用注入的方法，也可以像调用普通方法那样调用注入的方法。利用ExpandoMetaClass，还可以将方法注入到POJO和POGO中。因此，我们可以在所有的类中享受到动态能力。

然而，ExpandoMetaClass也有些限制。注入的方法只能从Groovy代码内调用，不能从编译过的Java代码中调用，也不能从Java代码中通过反射来使用。不过要从Java中调用它们，有一个变通方案，参见10.6节。

## 13.3 向具体的实例中注入方法

前面介绍了向类中动态注入方法的不同方式，也可以像向类中添加行为那样向具体的实例中添加行为。假设接收到一个Person，并且想基于某个条件或状态在它上面执行一些操作。在其上注入一组可复用的方法或工具函数应该会更容易；然而，我们不希望将这些操作全面地应用到所有Person上。Groovy使得向实例中注入方法相当简单。

每个实例都有一个MetaClass。如果希望一个实例的行为与从相同类初始化而来的其他对象不同，可以将方法注入到从这个具体的实例获得的metaClass中。也可以选择创建一个ExpandoMetaClass实例，将指定方法加入其中（包括我们想从这个实例当前的metaClass中保存下来的方法），对它进行初始化（用于说明方法/属性添加完成），将其附到想要增强的实例上。下面这个例子向一个Person实例中添加了一个方法：

**InjectionAndSynthesisWithMOP/InjectInstance.groovy**

```
class Person {
  def play() { println 'playing...' }
}
def emc = new ExpandoMetaClass(Person)
emc.sing = { ->
  'oh baby baby...'
}
emc.initialize()

def jack = new Person()
def paul = new Person()

jack.metaClass = emc

println jack.sing()

try {
  paul.sing()
} catch(ex) {
  println ex
}
```

前面代码的输出如下：

```
oh baby baby...
groovy.lang.MissingMethodException:
  No signature of method: Person.sing()
  is applicable for argument types: () values: []
Possible solutions: find(), find(groovy.lang.Closure),
  is(java.lang.Object), any(), print(java.lang.Object),
  print(java.io.PrintWriter)
```

通过设置jack上的MetaClass实例，我们为这为勇敢的朋友注入了sing()方法。现在可以在jack上调用sing()了。然而，如果想在另一个Person实例上调用该方法，则会失败。



现在成功地向jack添加了sing()，但是如果他的歌喉像我的一样，我们希望他只在洗手间唱。Groovy提供了一种方便的方式来从实例中去掉这些注入的方法——只需要将metaClass属性设置为null。

**InjectionAndSynthesisWithMOP/InjectInstance.groovy**

```
jack.metaClass = null
try {
  jack.play()
  jack.sing()
} catch(ex) {
  println ex
}
```

现在就去掉了已经添加的方法，从输出中可以看到，之后对这个方法的任何调用都将失败。只有注入的方法会受到影响，任何预先定义的方法（比如play()）仍然可以使用。

```
playing...
groovy.lang.MissingMethodException:
  No signature of method: Person.sing() is applicable ...
```

前面用了几个步骤，来创建ExpandoMetaClass、向其中加入方法以及初始化。其实不必这么麻烦。我们可以简单地将方法设置到该实例的metaClass属性上，如下所示：

**InjectionAndSynthesisWithMOP/InjectInstanceMetaClass.groovy**

```
class Person {
  def play() { println 'playing...' }
}
def jack = new Person()
def paul = new Person()

jack.metaClass.sing = { ->
  'oh baby baby...'
}
println jack.sing()
```

```
try {
  paul.sing()
} catch(ex) {
  println ex
}

jack.metaClass = null
try {
  jack.play()
  jack.sing()
} catch(ex) {
  println ex
}
```

这个版本的注入方法代码中少了很多噪音，而输出与之前的版本相同。

我们可以像注入`sing()`方法那样单独地向一个实例中注入多个方法，也可以像13.2节所做的那样，使用EMC DSL将方法分组。将方法分组的语法如下：

```
jack.metaClass {
  sing = { ->
    'oh baby baby...'
  }
  dance = { ->
    'start the music...'
  }
}
```

利用`ExpandoMetaClass`，可以向Groovy和Java类中注入方法。还可以像本节看到的那样，向具体的实例中注入方法。如果想向多个类注入一组方法，`ExpandoMetaClass`还通过Mixins提供的另一个便利之处，下一节将介绍。

## 13.4　使用 Mixin 注入方法

Java中可以实现多个接口，但只允许继承一个类。Groovy也遵守Java的这种语义，但是还支持灵活地将其他多个类中的实现拉入进来。

C++等语言中存在的一些与多重继承相关的问题[1]，促使Java决定对此加以限制。为了避免多个实现被拉到一起时可能出现的冲突和混乱，Groovy的做法是允许方法组合与合作，而不是冲突，本节将予以介绍。

Groovy的Mixin[2]是一种运行时能力，可以将多个类中的实现引入进来或混入。如果将一个类

① 多重继承中最典型的是菱形继承问题，无法确定所继承的功能到底来自哪个父类，所以C++又引入了复杂的虚拟继承机制。——译者注

② Mixin这个词由mix和in组合而来，顾名思义，即“混入进来”。翻译中，Mixin作为名词表示Groovy的这种特性时，保留不译；而作为动词表示Mixin的实现时，则根据表达的需要译为“混入”或“混入进来”。——译者注

混入到另一个类中，Groovy会在内存中把这些类的类实例链接起来。当调用一个方法时，Groovy首先将调用路由到混入的类中，如果该方法存在于这个类中，则在此处理。否则由主类处理。可以将多个类混入到一个类中，最后加入的Mixin优先级最高。

Mixin可以将一个类混入到多个类中，也可以将多个类混入到一个类中。为了便于学习，下面创建一个Friend类，并将其方法注入到几个类中。

**InjectionAndSynthesisWithMOP/mixin.groovy**

```
class Friend {
  def listen() {
    "$name is listening as a friend"
  }
}
```



Friend类看上去就是一个普通的类。它的listen()方法表示朋友的行为——好朋友总是愿意倾听啊。这个方法简单地打印name的值和一条很短的消息。name属性本身没有在这个类中做任何定义，它将由该类所混入的类提供。

下面就来考察Groovy为支持混入类提供的可选方案。

首先，可以使用@Mixin注解语法，将Mixin注入到Person类中（如下面代码所示）。作为一种选择，也可以使用静态初始化器将Mixin引入到这个类中，就像这样：class Person { static { mixin Friend} ...}。

**InjectionAndSynthesisWithMOP/mixin.groovy**

```
@Mixin(Friend)
class Person {
  String firstName
  String lastName
  String getName() { "$firstName $lastName"}
}

john = new Person(firstName: "John", lastName: "Smith")
println john.listen()
```

@Mixin注解会将作为参数提供的类中的方法添加到被注解的类中。在这个例子中，Friend的方法被添加到了Person中。通过向注解提供由多个类名组成的列表，可以混入多个类，就像这样：@Mixin([Friend, Teacher])。

如前面例子所示，可以在任何一个Person实例上调用使用Mixin注入的方法。下面输出说明了Person对混入的listen()方法的响应。

```
John Smith is listening as a friend
```

Mixin的语法非常优雅，而且简洁，但是注解本身限制了这种方式只能由类的作者使用。如果没有类的源代码，或者不想修改源代码，就不能使用这种方式。

向已有的类中注入行为，即可实现向Groovy和Java类中注入方法。没有源代码也可以实现混入。下面看一下在运行时动态实现混入的语法：狗是人类最好的朋友，我们就创建一个Dog类，然后将Friend混入到这个类中。

InjectionAndSynthesisWithMOP/mixin.groovy

```
class Dog {
  String name
}

Dog.mixin Friend

buddy = new Dog(name: "Buddy")
println buddy.listen()
```

这里没有使用注解，而是在Dog类上调用了mixin()方法，并将想混入到这个类中的类的名字传给了它。现在所有的Dog实例都可以使用混入的类中的方法了。下面输出演示了调用listen()方法的结果。

```
Buddy is listening as a friend
```

以上介绍了两种混入类的方式：使用注解和使用特殊的mixin()方法。也可以有选择地向一个类的具体实例中混入类。编写一个Cat类，看看这是如何工作的：

InjectionAndSynthesisWithMOP/mixin.groovy

```
class Cat {
  String name
}

try {
  rude = new Cat(name: "Rude")
  rude.listen()
} catch(ex) {
  println ex.message
}
```

命名为rude的Cat实例并不支持Friend的任何方法，因为猫通常并不友好，至少不像我们期待的那样友好。因此，当尝试在rude上调用listen()方法时，出现了灾难性的结果：

```
No signature of method: Cat.listen() is applicable
  for argument types: () values: []
Possible solutions: ...
```

并非所有的猫都是如此，可以根据意愿从基因方面修改具体的实例。下面创建另一个Cat实例——socks，它有点特殊[①]，通过在它的metaClass上调用mixin()方法，混入了Friend的行为。

① 美国前总统克林顿当政时期的白宫第一猫“袜子”。——编者注

InjectionAndSynthesisWithMOP/mixin.groovy

```
socks = new Cat(name: "Socks")
socks.metaClass.mixin Friend
println socks.listen()
```

新得到的朋友socks对着listen()调用喵喵地叫了起来。

```
Socks is listening as a friend
```

前面介绍了如何向类和具体的实例混入行为，还可以混入多个行为，下一节将介绍。

## 13.5 在类中使用多个 Mixin



当混入多个类时，所有这些类的方法在目标类中都是可用的。默认情况下，方法会因为冲突而被隐藏。换言之，当作为Mixin的两个或多个类中存在名字相同、参数签名也相同的方法时，最后加入到Mixin中的方法会隐藏掉已经注入的方法。

通过编程实现，以方法调用链条的方式将这些方法链接起来，反而可以让它们合作。创建一个过滤器链，对这些方法接受的参数加以变换，下面将看到Mixin是如何提供这样的设计选择的。

一个应用需要不同类型的输出目标，可能是文件、套接字、Web服务和简单的字符串，等等。将这些一般化为一个抽象基类——Writer。

InjectionAndSynthesisWithMOP/Filters.groovy

```
abstract class Writer {
  abstract void write(String message)
}
```

StringWriter是这个类的一个具体的实现，它会在write()方法中将给定消息写入到一个StringBuilder中。

InjectionAndSynthesisWithMOP/Filters.groovy

```
class StringWriter extends Writer {
  def target = new StringBuilder()

  void write(String message) {
    target.append(message)
  }

  String toString() { target.toString() }
}
```

可以根据自己的意愿编写其他的Writer实现。比如利用如下方法，使用已经创建的StringWriter写入一些东西：

**InjectionAndSynthesisWithMOP/Filters.groovy**

```
def writeStuff(writer) {
  writer.write("This is stupid")
  println writer
}

def create(theWriter, Object[] filters = []) {
  def instance = theWriter.newInstance()
  filters.each { filter -> instance.metaClass.mixin filter }
  instance
}
writeStuff(create(StringWriter))
```

writeStuff()方法接受一个Writer实例，使用write()方法写入一条有智能的消息，然后打印Writer中保存的内容。对于作为第一个参数提供的Writer的一个具体的派生类，使用create()方法来创建其实例。之后将一个由类组成的可选的列表混入到这个实例中，并返回完成混入的实例。

下面是未混入任何行为的情况下，调用writeStuff()方法的结果：

```
This is stupid
```

现在的代码只不过是将给定消息写入到目标，比如StringWriter，或是可以实现的其他特化的Writer。在此期间，需求发生了变化，要求将给定参数的值以大写形式写出。

我们不想修改任何具体的写入目标，比如StringWriter或基类Writer。要求以大写形式打印，可能只是一个先兆，后面可能还会有很多这样的变换或过滤需求；如果对这些类做出任何修改，当这样的请求出现时，它们可能会无法扩展。

下面创建一个独立的UpperCaseFilter类，来看看Mixin对于实现灵活的设计有何帮助：

**InjectionAndSynthesisWithMOP/Filters.groovy**

```
class UppercaseFilter {

  void write(String message) {
    def allUpper = message.toUpperCase()

    invokeOnPreviousMixin(metaClass, "write", allUpper)
  }
}
```

UpperCaseFilter的write()方法将给定的参数message转换成了全部大写，然后调用了一个尚未编写的invokeOnPreviousMixin()方法。这个write()方法聚焦于它的一点责任——过滤和转换消息。之后将修改后的消息传递给Mixin链条左侧的下一个对象或过滤器。

现在需要实现invokeOnPreviousMixin()方法。可以将其编写为一个独立的方法，也可以将其注入到Object基类中。后者的优势是，该方法会作为一个实例方法存在于任何类上。在这

个方法中，需要为所处理的实例取到其Mixin列表中的前一个Mixin。

mixin()方法用于向一个类或实例中添加一个或多个Mixin。Groovy提供了一个名为mixedIn的属性，用于为实例保存有序的Mixin列表。

请记住，所添加的Mixin构成了一个链条，通向被混入的目标实例。遍历Mixin组成的链条，可以找到新添加的Mixin，或是找到位于列表前面的最终的目标实例，如下所示：

**InjectionAndSynthesisWithMOP/Filters.groovy**

```
Object.metaClass.invokeOnPreviousMixin = {
  MetaClass currentMixinMetaClass, String method, Object[] args ->
  def previousMixin = delegate.getClass()
  for(mixin in mixedIn.mixinClasses) {
    if(mixin.mixinClass.theClass ==
      currentMixinMetaClass.delegate.theClass) break
    previousMixin = mixin.mixinClass.theClass
  }
  mixedIn[previousMixin]."$method"(*args)
}
```

链条中最左侧的实例的类型就是目标实例，可以使用delegate.getClass()获得。之后遍历保存在mixedIn这个LinkedHashSet中的类的链表，直到找到当前Mixin的前一个Mixin。最后，在一个Mixin的上下文中，调用这个Mixin或是位于链表前面的目标实例上的给定方法。

要看一下这些努力的成果，在一个混入了UpperCaseFilter的StringWriter实例上调用writeStuff()方法。

**InjectionAndSynthesisWithMOP/Filters.groovy**

```
writeStuff(create (StringWriter, UppercaseFilter))
```

前面的调用向writeStuff()方法发送了一个StringWriter实例，其中链接了一个UpperCaseFilter实例，如图13-1所示。

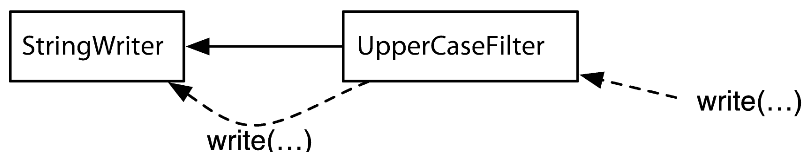



图13-1 将UpperCaseFilter Mixin链接到一个StringWriter实例

在这个StringWriter实例上，对write()方法的调用首先被路由到所链接的UpperCaseFilter。UpperCaseFilter上的write()方法对给定的参数加以变换，再将调用转发到目标实例上，如下所示：

```
THIS IS STUPID
```

辛苦想出的这种可扩展的设计很快就派上了用场——我们又被要求过滤出消息中的脏话。去掉脏话这个任务非常艰巨，所以第一次只是去掉了stupid——负责任的父母会经常告诫自己的孩子别说这个词。

**InjectionAndSynthesisWithMOP/Filters.groovy**

```
class ProfanityFilter {
  void write(String message) {
    def filtered = message.replaceAll('stupid', 's*****')
    invokeOnPreviousMixin(metaClass, "write", filtered)
  }
}

writeStuff(create (StringWriter, ProfanityFilter))
```

ProfanityFilter的write()方法替换掉了所有以小写形式出现的这个具有冒犯性的词，然后将修改后的消息转发给链条中位于左侧的实例。目标会接收到相应的过滤后的消息，如下面输出所示：

```
This is s*****
```

下面例子按顺序链接起了前面编写的两种过滤器，使用Mixin的这种设计所带来的灵活性和可扩展性，在这个例子中大放异彩。

**InjectionAndSynthesisWithMOP/Filters.groovy**

```
writeStuff(create(StringWriter, UppercaseFilter, ProfanityFilter))
writeStuff(create(StringWriter, ProfanityFilter, UppercaseFilter))
```

第一个调用中创建了一个链条，UpperCaseFilter后面跟着ProfanityFilter；第二个调用中将这两个过滤器的顺序反了过来，如图13-2所示。

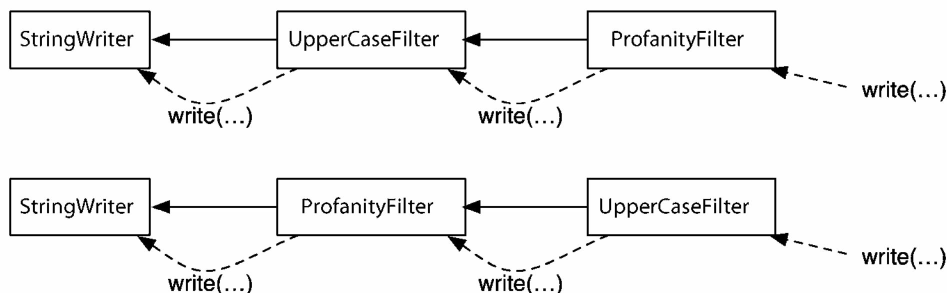



图13-2　将UpperCaseFilter和ProfanityFilter两个过滤器以不同顺序链接到一个StringWriter实例中

混入行为的顺序至关重要。方法调用会向链条中的左侧传递，如输出所示：

```
THIS IS S*****
THIS IS STUPID
```

因为这个`ProfanityFilter`非常简单，只查找小写单词，当链条上的第一个过滤器是`UpperCaseFilter`时，单词会被转换成大写，所以脏话没有过滤出去。可以使用这种技巧来观察执行顺序。

这一节中我们看到了Mixin的强大，了解了如何在类和实例的层次创建它们，也知道了如何将其链接起来以实现可扩展的设计。如果我们是设计模式的拥趸，会看出这是装饰模式（Decorator Pattern）的一个实现。向类中动态注入方法非常强大，但是利用下一章即将介绍的方法合成会使Groovy元编程的灵活性提升一个层次。

本章介绍了如何拦截和注入方法。Groovy的MOP使执行类AOP的行为非常容易，既可以创建高度动态的代码，也可以创建只有几行的高度可复用代码。下一章将在动态行为上更上一层，介绍如何动态合成或生成方法。

第 14 章

# MOP方法合成

14

添加行为可分为两类：注入和合成。

*方法注入*（method injection）用来指代这种情况：编写代码时知道想要添加到一个或多个类中的方法的名字。利用方法注入，可以动态地向类中添加行为。也可以向任意数目的类中注入一组实现某一特定功能的可复用方法，就像工具函数。可以通过使用分类，使用`ExpandoMetaClass`或Groovy的Mixin工具来注入方法。第13章介绍了这些技术。

*方法合成*（method synthesis）所指的情况是：想在调用时动态地确定方法的行为。Groovy的`invokeMethod()`、`methodMissing()`和`GroovyInterceptable`对于方法合成非常有用。比如，Grails/GORM会在调用时为领域对象合成像`findByFirstName()`和`findByFirstNameAndLastName()`这样的查找方法。

合成的方法可能直到调用时才会作为独立的方法存在。当调用了一个不存在的方法时，Groovy可以拦截该调用。我们可以在这里直接实现相应的方法，并将其缓存下来供后续调用使用，然后调用这个实现。Grails的创建者Graeme Rocher称其为“拦截、缓存、调用”模式。

这一章将介绍如何向类和具体的实例添加动态的方法。不同于在编译时提供一组指定好的方法（和行为），可以让类基于其在应用中的执行路径，或基于其当前状态来引入行为。这样对象看上去会很活跃、很智能，可以随着演进吸收新行为。本章学习的技术有助于理解动态方法在Groovy相关的工具中是如何使用的。比如，这些技术有助于看清持久化对象（GORM）在Grails中是如何实现的，以及像Gradle这样的构建工具又是如何利用那么点代码实现如此灵活、动态的行为的。

## 14.1 使用 methodMissing 合成方法

现在，我们已经掌握了向类或实例中注入具体方法的方式。本节将使用灵活和动态的名字合成方法。我们没有提前确定这些名字。实际上，可以让类的用户决定这些名字，只要遵循预先设定的约定即可。当他们调用了一个不存在的方法时，可以将调用拦截下来，然后直接创建一个实现。这种实现是量身定制的。换句话说，只有用户要求的时候才会创建。

Grails/GORM中为领域类实现了方法合成。假设一个领域类（这个类表示的是要持久化到一

个数据库表中的信息）Person有几个字段（对应表中的列），比如firstName、lastName和cityOfResidence等，可能随时会加入其他字段。GORM允许Person类的用户调用诸如findByFirstName()、findByLastName()和findByFirstNameAndLastName()这样的方法，甚至如果age也是Person的一个字段，还可以调用findByFirstNameAndAge()。Person类不会提前创建其中任何一个方法。每个方法都是在运行时第一次被调用的时候合成的。本章接下来将探讨在Groovy中如何合成方法。

在Groovy中，通过实现methodMissing()，可以拦截对不存在的方法的调用。同样，通过实现propertyMissing()，也可以拦截对不存在的属性的访问。在这些方法内，可以动态地为这些不存在的方法或属性实现相应逻辑。我们会基于所定义的约定推断其语义。比如，以find打头的方法名可能意味着查询，而以update打头的方法名则可能意味着保存，依此类推。

下面看一个合成方法的例子。jack是一个无聊的人（Person），只工作不玩耍，我们想把他变成一个多项全能运动员，能够玩各种各样的运动。

14

**InjectionAndSynthesisWithMOP/MethodSynthesisUsingMethodMissing.groovy**

```
class Person {
  def work() { "working..." }

  def plays = ['Tennis', 'VolleyBall', 'BasketBall']

  def methodMissing(String name, args) {
    System.out.println "methodMissing called for $name"
    def methodInList = plays.find { it == name.split('play')[1]}
    if (methodInList) {
      "playing ${name.split('play')[1]}..."
    } else {
      throw new MissingMethodException(name, Person.class, args)
    }
  }
}

jack = new Person()

println jack.work()
println jack.playTennis()
println jack.playBasketBall()
println jack.playVolleyBall()
println jack.playTennis()

try {
  jack.playPolitics()
} catch(Exception ex) {
  println "Error: " + ex
}
```

这段代码的输出如下：

```
working...
methodMissing called for playTennis
playing Tennis...
methodMissing called for playBasketBall
playing BasketBall...
methodMissing called for playVolleyBall
playing VolleyBall...
methodMissing called for playTennis
playing Tennis...
methodMissing called for playPolitics
Error: groovy.lang.MissingMethodException:
  No signature of method: Person.playPolitics()
  is applicable for argument types: () values: []
```

work()是Person上预先定义的唯一一个领域方法。调用work()会直接到达该方法。然而，如果调用的是不存在的方法，则会被路由到methodMissing()方法。在methodMissing()中，如果被调用的方法以play开头，并以plays数组中的一个名字结尾，就接受它。而且可以动态地修改plays数组，以添加我们想要的其他运动，给人留下jack又习得了新技能的印象。如果被调用的方法不被支持的，比如playPolitics()，就抛出MissingMethodException。

从调用者的视角看，调用普通的方法和合成的方法并无二致。

前面的实现极为动态，但是存在一个陷阱。重复调用一个不存在的方法，比如playTennis()，每次处理都会带来同样的性能问题。在第一次调用时注入该方法可以提高效率。再次说明，Graeme Rocher称这种技术为“拦截、缓存、调用”模式。我们将在第一次调用时合成方法，将其注入到MetaClass中缓存下来，然后调用这一注入的方法。代码如下：

**InjectionAndSynthesisWithMOP/MethodSynthesisUsingMethodMissing2.groovy**

```
class Person {
  def work() { "working..." }

  def plays = ['Tennis', 'VolleyBall', 'BasketBall']

  def methodMissing(String name, args) {
    System.out.println "methodMissing called for $name"
    def methodInList = plays.find { it == name.split('play')[1]}

    if (methodInList) {
      def impl = { Object[] vargs ->
          "playing ${name.split('play')[1]}..."
      }

      Person instance = this
      instance.metaClass."$name" = impl //以后再调用就会使用它
```

```
      impl(args)
    } else {
      throw new MissingMethodException(name, Person.class, args)
    }
  }
}

jack = new Person()
println jack.playTennis()
println jack.playTennis()
```

前面代码的输出说明，合成的方法在第一次调用时被缓存了下来：

```
methodMissing called for playTennis
playing Tennis...
playing Tennis...
```

methodMissing()方法仅在第一次调用所支持的不存在方法时会被调用到。第二次调用（以及随后的调用）同一方法时，会直接抵达向实例的MetaClass中注入的实现（闭包）。



**methodMissing与GroovyInterceptable**

对于实现了GroovyInterceptable的对象，调用该对象上的任何方法，都会调用到invokeMethod()。与invokeMethod()不同的是，只有调用不存在的方法时，才会调用到methodMissing()。如果一个对象实现了GroovyInterceptable，不管被调用的方法是否存在，invokeMethod()都会被调用（如果存在的话）。只有对象将控制转移给其MetaClass的invokeMethod()时，methodMissing()才会被调用。

12.2节中使用了GroovyInterceptable来拦截方法调用。也可以将其与methodMissing()混合使用，来拦截对现有的方法和合成的方法的调用，如下所示：

**InjectionAndSynthesisWithMOP/InterceptingMissingMethods.groovy**

```
class Person implements GroovyInterceptable {
  def work() { "working..." }
  def plays = ['Tennis', 'VolleyBall', 'BasketBall']
  def invokeMethod(String name, args) {
    System.out.println "intercepting call for $name"

    def method = metaClass.getMetaMethod(name, args)

    if (method) {
      method.invoke(this, args)
    } else {
      metaClass.invokeMethod(this, name, args)
    }
```

```
  }

  def methodMissing(String name, args) {
    System.out.println "methodMissing called for $name"
    def methodInList = plays.find { it == name.split('play')[1]}

    if (methodInList) {
      def impl = { Object[] vargs ->
          "playing ${name.split('play')[1]}..."
        }

      Person instance = this
      instance.metaClass."$name" = impl //以后再调用就会使用它

      impl(args)
    } else {
      throw new MissingMethodException(name, Person.class, args)
    }
  }
}
jack = new Person()
println jack.work()
println jack.playTennis()
println jack.playTennis()
```

这段代码的输出如下：

```
intercepting call for work
working...
intercepting call for playTennis
methodMissing called for playTennis
playing Tennis...
intercepting call for playTennis
playing Tennis...
```

方法合成是Groovy最强大的特性之一。该特性在基于Groovy的库和框架中应用广泛，比如easyb和GORM。我也经常用到该特性，主要是为复杂的业务逻辑处理编写可扩展的代码，一般只需要几行代码。

## 14.2　使用 ExpandoMetaClass 合成方法

上一节介绍了如何合成方法。然而，如果我们无权编辑类的源文件，或者该类并非一个POGO，那种方法就行不通了。对于这类情况，可以使用ExpandoMetaClass来合成方法。

12.2节中讲过如何与MetaClass交互。不同于为每个领域方法提供一个拦截器，这里将在MetaClass上实现methodMissing()方法。仍以上一节的Person类（还有无聊的jack）为例，使用ExpandoMetaClass，如下所示：

**InjectionAndSynthesisWithMOP/MethodSynthesisUsingEMC.groovy**

```
class Person {
  def work() { "working..." }
}

Person.metaClass.methodMissing = { String name, args ->
  def plays = ['Tennis', 'VolleyBall', 'BasketBall']

  System.out.println "methodMissing called for $name"
  def methodInList = plays.find { it == name.split('play')[1]}

  if (methodInList) {
    def impl = { Object[] vargs ->
        "playing ${name.split('play')[1]}..."
    }
    Person.metaClass."$name" = impl //以后再调用就会使用它
    impl(args)
  } else {
    throw new MissingMethodException(name, Person.class, args)
  }
}
jack = new Person()
println jack.work()
println jack.playTennis()
println jack.playTennis()

try {
  jack.playPolitics()
} catch(ex) {
  println ex
}
```

前面代码的输出如下：

```
working...
methodMissing called for playTennis
playing Tennis...
playing Tennis...
methodMissing called for playPolitics
groovy.lang.MissingMethodException:
  No signature of method: Person.playPolitics()
  is applicable for argument types: () values: []
```

如果我们的类中也提供了methodMissing()方法，MetaClass的methodMissing()方法会优先被调用。类的MetaClass上的方法会覆盖掉类中的方法。

当在jack上调用work()时，Person的work()被直接执行。然而如果调用的是不存在的方法，则调用会被路由到Person的MetaClass中的methodMissing()。这个方法实现了与上一节的解决方案类似的逻辑。重复调用所支持的不存在方法不会产生开销，这在前面第二次调用playTennis()的输出中可以看到。因为第一次调用的实现已经被缓存了。

12.2节中还使用ExpandoMetaClass的invokeMethod()拦截了方法调用。我们可以混合使用invokeMethod()和methodMissing()，来拦截对现有的方法和合成的方法的调用，如下所示：

InjectionAndSynthesisWithMOP/MethodSynthesisAndInterceptionUsingEMC.groovy

```
class Person {
  def work() { "working..." }
}
Person.metaClass.invokeMethod = { String name, args ->
  System.out.println "intercepting call for ${name}"

  def method = Person.metaClass.getMetaMethod(name, args)

  if (method) {
    method.invoke(delegate, args)
  } else {
    Person.metaClass.invokeMissingMethod(delegate, name, args)
  }
}

Person.metaClass.methodMissing = { String name, args ->
  def plays = ['Tennis', 'VolleyBall', 'BasketBall']

  System.out.println "methodMissing called for ${name}"
  def methodInList = plays.find { it == name.split('play')[1]}

  if (methodInList) {
    def impl = { Object[] vargs ->
        "playing ${name.split('play')[1]}..."
    }

    Person.metaClass."$name" = impl //以后再调用就会使用它

    impl(args)
  } else {
    throw new MissingMethodException(name, Person.class, args)
  }
}

jack = new Person()
println jack.work()
println jack.playTennis()
println jack.playTennis()
```

前面代码的输出如下：

```
intercepting call for work
working...
intercepting call for playTennis
methodMissing called for playTennis
```

```
playing Tennis...
intercepting call for playTennis
playing Tennis...
```

**invokeMethod与methodMissing的对比**

invokeMethod()是GroovyObject的一个方法。methodMissing()则是Groovy中后来引入的，是基于MetaClass的方法处理的组成部分。如果目标是处理对不存的方法的调用，应该实现methodMissing()，因为它的开销较低。如果目标是拦截所有的方法调用，而不管方法存在与否，则应使用invokeMethod()。

## 14.3　为具体的实例合成方法



在13.3节中，我们知道了如何向某个类的具体实例中注入方法。通过向具体的实例提供专用的MetaClass，也可以将方法合成到这些实例中。下面是一个例子：

**InjectionAndSynthesisWithMOP/SynthesizeInstance.groovy**

```
class Person {}

def emc = new ExpandoMetaClass(Person)
emc.methodMissing = { String name, args ->
  "I'm Jack of all trades... I can $name"
}
emc.initialize()

def jack = new Person()
def paul = new Person()

jack.metaClass = emc

println jack.sing()
println jack.dance()
println jack.juggle()

try {
  paul.sing()
} catch(ex) {
  println ex
}
```

这段代码的输出如下：

```
I'm Jack of all trades... I can sing
I'm Jack of all trades... I can dance
I'm Jack of all trades... I can juggle
```

```
groovy.lang.MissingMethodException:
  No signature of method: Person.sing()
  is applicable for argument types: () values: []
```

能够在实例的层次合成方法，非常有用。在测试中，或者在一个Web应用的特定Web请求中，我们可以修改一个选定实例的行为，而不影响Java虚拟机中该实例所关联的类.

还有一点非常强大，就是能够基于实例的当前状态或实例所接受的输入创建动态的方法或行为。这为创建和实现高度动态的DSL铺平了道路，我们将在后面看到。

本章介绍了如何合成方法。下一章将探讨如何动态地创建类，并且感受一下如何在各种元编程技术中做出选择。

第 15 章

# MOP技术汇总

15

上一章介绍了如何合成方法，本章探讨如何合成一个完整的类。不同于提前创建显式的类，可以在运行时创建类，这样更灵活。虽然委托优于继承，但是委托在Java中并不好实现，而在本章中将看到，Groovy的MOP允许仅使用一行代码实现方法委托。本章最后将回顾前几章介绍的MOP技术。[①]

## 15.1 使用 Expando 创建动态类



Groovy中可以完全在运行时创建一个类。假设要构建一个用于配置设备的应用，而我们对这些设备一无所知，只知道它们有些属性和配置脚本，那就无法在编写代码时奢侈地为每个设备创建一个类。因此，需要在运行时合成与这些设备打交道并完成配置的类。在Groovy中，类可以根据命令在运行时产生。

Groovy的Expando类提供了动态合成类的能力，也因其动态可扩展性而得名。可以在构建时使用一个Map为其指定属性和方法，也可以动态地随时指定。下面就从一个例子入手，合成一个Car类。这里会介绍两种使用Expando创建这个类的方法。

**MOPpingUp/UsingExpando.groovy**

```
carA = new Expando()
carB = new Expando(year: 2012, miles: 0)
carA.year = 2012
carA.miles = 10

println "carA: " + carA
println "carB: " + carB
```

输出如下：

```
carA: {year=2012, miles=10}
carB: {year=2012, miles=0}
```

---

① 本章英文标题为“MOPping Up”，有双关之意。mop up本身有“清理、清扫”之意，MOP本身又是本书第三部分所探讨的“元对象协议”。——译者注

此处创建的第一个Expando实例——carA，没有任何属性或方法。之后向该实例中注入了year和miles属性。而创建的第二个Expando实例——carB，在构建时提供了初始化过的year和miles属性。

我们不仅可以定义属性，还可以定义方法，并像调用任何方法那样调用它们。下面来试一下。再次重申，我们既可以在构建时定义方法，也可以以后随意注入：

**MOPpingUp/UsingExpando.groovy**

```
car = new Expando(year: 2012, miles: 0, turn: { println 'turning...' })
car.drive = {
  miles += 10
  println "$miles miles driven"
}

car.drive()
car.turn()
```

这段代码的输出如下：

```
10 miles driven
turning...
```

假设有一个输入文件，其中保存了一些汽车用的数据，如下所示：

**MOPpingUp/car.dat**

```
miles, year, make
42451, 2003, Acura
24031, 2003, Chevy
14233, 2006, Honda
```

无需显式创建一个Car类，就能方便地使用Car对象（如下列代码所示）。解析car.dat文件的内容，首先提取出属性名。然后为输入文件中的每行数据创建一个Expando实例，并使用行中的属性值填充这些实例。甚至还以闭包的形式添加了一个方法，以计算截止到2008年，汽车每年驾驶的平均公里数。一旦对象创建出来，就可以动态地访问其属性或调用其方法了。也可以通过名字来使用方法或属性，下列代码的最后有演示。

**MOPpingUp/DynamicObjectsUsingExpando.groovy**

```
data = new File('car.dat').readLines()

props = data[0].split(", ")
data -= data[0]

def averageMilesDrivenPerYear = { miles.toLong() / (2008 - year.toLong()) }

cars = data.collect {
  car = new Expando()
  it.split(", ").eachWithIndex { value, index ->
```

```
    car[props[index]] = value
  }

  car.ampy = averageMilesDrivenPerYear

  car
}

props.each { name -> print "$name " }
println " Avg. MPY"

ampyMethod = 'ampy'
cars.each { car ->
  for(String property : props) { print "${car[property]} " }
  println car."$ampyMethod"()
}

// 你也可能想通过名字访问属性或方法
car = cars[0]
println "$car.miles $car.year $car.make ${car.ampy()}"
```

这段代码的输出如下：

```
miles year make  Avg. MPY
42451 2003 Acura 8490.2
24031 2003 Chevy 4806.2
14233 2006 Honda 7116.5
42451 2003 Acura 8490.2
```



想在运行时合成类时，可以使用`Expando`。它是轻量级的，而且非常灵活。比如，当为单元测试创建模拟（Mock）对象时，该特性会大放光彩，18.8节将予以介绍。

## 15.2　方法委托：汇总练习

继承用来扩展一个类的行为，而委托依赖所包含或聚合的对象可以提供一个类的行为。如果想用一个对象替代另一个对象，应该选择继承；如果只是想简单地使用一个对象，则应该选择委托。请将继承保留给`is-a`或`kind-of`关系；大多数情况下，应该首选委托（参见*Effective Java* [Blo08]）。然而使用继承编写程序很容易，只需要一个`extends`关键字。委托编写起来就困难了，因为必须编写很多方法，用以将调用路由给所包含对象的方法。Groovy可以帮助我们做正确的事。通过使用MOP，用一行代码即可轻松实现委托，本节将会介绍。

在下面的例子中，一个`Manager`想把工作委托给一个`Worker`或一个`Expert`。使用`methodMissing()`和`ExpandoMetaClass`来实现该功能。如果在`Manager`上调用的一个方法并不存在，该实例的`methodMissing()`方法会将调用路由给`Worker`或`Expert`，只要其中有一个能够成功处理`respondsTo()`方法即可（参见11.2节）。如果这些委托对象都不能处理某个方法，则

Manager无法处理该方法。

**MOPpingUp/Delegation.groovy**

```
class Worker {
  def simpleWork1(spec) { println "worker does work1 with spec $spec" }
  def simpleWork2() { println "worker does work2" }
}

class Expert {
  def advancedWork1(spec) { println "Expert does work1 with spec $spec" }
  def advancedWork2(scope, spec) {
    println "Expert does work2 with scope $scope spec $spec"
  }
}

class Manager {
  def worker = new Worker()
  def expert = new Expert()
  def schedule() { println "Scheduling ..." }
  def methodMissing(String name, args) {
    println "intercepting call to $name..."
    def delegateTo = null

    if(name.startsWith('simple')) { delegateTo = worker }
    if(name.startsWith('advanced')) { delegateTo = expert }
    if (delegateTo?.metaClass.respondsTo(delegateTo, name, args)) {
      Manager instance = this
      instance.metaClass."${name}" = { Object[] varArgs ->
            delegateTo.invokeMethod(name, varArgs)
      }
      delegateTo.invokeMethod(name, args)
    } else {
      throw new MissingMethodException(name, Manager.class, args)
    }
  }
}
peter = new Manager()
peter.schedule()
peter.simpleWork1('fast')
peter.simpleWork1('quality')
peter.simpleWork2()
peter.simpleWork2()
peter.advancedWork1('fast')
peter.advancedWork1('quality')
peter.advancedWork2('prototype', 'fast')
peter.advancedWork2('product', 'quality')
try {
```

```
  peter.simpleWork3()
} catch(Exception ex) {
  println ex
}
```

输出如下：

```
Scheduling ...
intercepting call to simpleWork1...
worker does work1 with spec fast
worker does work1 with spec quality
intercepting call to simpleWork2...
worker does work2
worker does work2
intercepting call to advancedWork1...
Expert does work1 with spec fast
Expert does work1 with spec quality
intercepting call to advancedWork2...
Expert does work2 with scope prototype spec fast
Expert does work2 with scope product spec quality
intercepting call to simpleWork3...
groovy.lang.MissingMethodException:
  No signature of method: Manager.simpleWork3()
  is applicable for argument types: () values: []
```



前面实现了一种委托调用的方式，但是工作量非常大。我们不希望每次想使用委托的时候都花上这么大的力气。可以重构这段代码，以便复用。先来看看当重构之后的代码用于Manager类中时，看上去是什么样子的：

**MOPpingUp/DelegationRefactored.groovy**

```
class Manager {
  { delegateCallsTo Worker, Expert, GregorianCalendar }

  def schedule() { println "Scheduling ..." }
}
```

这段代码短小、漂亮。初始化块中调用了一个尚待实现的方法delegateCallsTo()，并将想用于委托未实现方法的类的名字传给了它。如果想在另一个类中使用委托，现在只需要拿走初始化块中的这行代码。下面看看精巧的delegateCallsTo()方法：

**MOPpingUp/DelegationRefactored.groovy**

```
Object.metaClass.delegateCallsTo = {Class... klassOfDelegates ->
  def objectOfDelegates = klassOfDelegates.collect { it.newInstance() }
  delegate.metaClass.methodMissing = { String name, args ->
      println "intercepting call to $name..."
      def delegateTo = objectOfDelegates.find {
                    it.metaClass.respondsTo(it, name, args) }
      if (delegateTo) {
```

```
        delegate.metaClass."${name}" = { Object[] varArgs ->
          delegateTo.invokeMethod(name, varArgs)
        }
        delegateTo.invokeMethod(name, args)
      } else {
        throw new MissingMethodException(name, delegate.getClass(), args)
      }
    }
}
```

当在类的实例初始化器内调用delegateCallsTo()方法时，该方法会向类中添加一个methodMissing()方法。这个类在闭包内被称作delegate。该闭包会接受作为一个参数提供给delegateCallsTo()的Class列表，并创建一个由用于委托的类组成的列表，这些类负责实现委托方法。在methodMissing()中，调用会被路由给这些类中可以对所调用方法做出响应的那个。如果这些类都无法响应，则调用失败。在提供给delegateCallsTo()方法的类的列表中，类出现的先后顺序也代表了其优先级：第一个优先级最高。当然，我们肯定要实际看看效果，运行如下代码，测试前面编写的Manager：

**MOPpingUp/DelegationRefactored.groovy**

```
peter = new Manager()
peter.schedule()
peter.simpleWork1('fast')
peter.simpleWork1('quality')
peter.simpleWork2()
peter.simpleWork2()
peter.advancedWork1('fast')
peter.advancedWork1('quality')
peter.advancedWork2('prototype', 'fast')
peter.advancedWork2('product', 'quality')
println "Is 2008 a leap year? " + peter.isLeapYear(2008)
try {
  peter.simpleWork3()
} catch(Exception ex) {
  println ex
}
```

输出如下：

```
Scheduling ...
intercepting call to simpleWork1...
worker does work1 with spec fast
worker does work1 with spec quality
intercepting call to simpleWork2...
worker does work2
worker does work2
intercepting call to advancedWork1...
Expert does work1 with spec fast
Expert does work1 with spec quality
```

```
intercepting call to advancedWork2...
Expert does work2 with scope prototype spec fast
Expert does work2 with scope product spec quality
intercepting call to isLeapYear...
Is 2008 a leap year? true
intercepting call to simpleWork3...
groovy.lang.MissingMethodException:
  No signature of method: Manager.simpleWork3()
  is applicable for argument types: () values: []
```

可以基于这个想法实现我们的需求。比如，如果想混入一些已经创建好的对象，可以将其作为一个数组发送给`delegateCallsTo()`的第一个参数，然后将它们与委托类中创建的对象一起使用。前面的例子演示了如何使用Groovy的MOP来实现诸如方法委托这样的动态行为。

从本节的例子中可以学到如何在运行时动态地修改实例的行为。如果喜欢，可以基于对象的当前状态修改委托。如果委托是静态的，则可以提前确定，没必要在运行时修改。这种情况可以简单地使用2.10.2节介绍的`@Delegate`注解（这是一种编译时元编程技术）。

## 15.3 MOP 技术回顾



第三部分介绍了很多可用于拦截、注入和合成方法的选项。本节要解决的问题是，了解哪个选项适合我们。

### 15.3.1 用于方法拦截的选项

第12章和13.1节探讨了方法拦截，我们可以使用`GroovyInterceptable`、`Expando MetaClass`或分类。

如果有权修改类的源代码，可以在想要拦截方法调用的类上实现`GroovyInterceptable`接口。做起来像实现`invokeMethod()`方法一样简单。

如果无法修改类，或者那是个Java类，则可以使用`ExpandoMetaClass`或分类。`ExpandoMetaClass`显然非常适合这种情况，因为一个`invokeMethod()`方法就可以负责拦截类上的任何方法。而分类则需要为每个要拦截的方法提供一个单独的方法。此外，如果使用分类，就必须使用`use()`块。

### 15.3.2 用于方法注入的选项

13.1节探讨了方法注入，可以使用分类或`ExpandoMetaClass`来实现。

在方法注入方面，分类完全可以与`ExpandoMetaClass`相媲美。如果使用分类，可以控制注入方法的位置。通过使用不同的分类，可以轻松实现不同版本的方法注入。我们还可以轻松地嵌

入和混用多个分类。而分类所提供的控制——即方法注入尽在use()块内起作用，而且被限制在执行线程上，也可以认为是一个限制。如果想在任意位置使用注入的方法，或者想注入静态方法和构造器，ExpandoMetaClass是更好的选择。不过还请注意，ExpandoMetaClass不是Groovy中的默认MetaClass。

借助ExpandoMetaClass，可以向某个类的具体实例注入方法，而不影响整个类。

### 15.3.3　用于方法合成的选项

14.1节探讨了方法合成，我们可以在Groovy对象或ExpandoMetaClass上使用method Missing()。

如果有权修改类的源代码，能够在类上为想要合成的方法实现methodMissing()方法，可以通过在第一次调用时注入方法来改进性能。如果同时需要拦截方法，实现GroovyInterceptable接口即可。

如果无法修改类，或者那是个Java类，可以将methodMissing()方法添加到类的ExpandoMetaClass中。如果同时想拦截方法，可以在ExpandoMetaClass上实现invokeMethod()。

借助ExpandoMetaClass，可以将方法合成到某个类的具体实例中，而不影响整个类。

元编程是Groovy得以大放异彩的一个关键特性。它使在运行时扩展程序，以及利用该语言的动态能力成为现实。到目前为止所学到的元编程技术，为我们提供了在运行时修改程序行为的能力。Groovy还支持在编译时修改程序的行为。下一章将探讨编译时元编程。

第 16 章

# 应用编译时元编程

与某些只有运行时能力的动态类型语言不同，Groovy同时提供了运行时和编译时元编程。

利用前面几章研究的运行时元编程，可以将与类和实例上方法的拦截、注入甚至合成相关的决策推迟到运行时。就元编程而言，大部分情况下，有这些就够了。编译时元编程是一种高级特性，可用于某些特殊情况，现在主要是框架或工具的编写者使用。

借助Groovy，可以在编译时分析和修改程序的结构。这可以为应用带来高度的可扩展行，同时支持添加新的横切特性。比如，无需修改源代码，即可为线程安全、日志消息和在代码的不同部分执行前置和后置检验操作等，对类进行检查。

编译时元编程，正是某些强大的特性和基于Groovy的工具背后的魔力所在。比如，Groovy 2中的类型检查器就实现为了一个抽象语法树（AST）变换。优雅的单元测试工具Spock使用这种方式来支持流畅的测试用例[①]。用于探测潜在的错误和代码不良味道的代码分析工具CodeNarc也大量使用了该特性[②]。该特性支撑起了Groovy的注解，比如2.10节介绍的`@Delegate`和`@Immutable`。

本章将介绍如何使用编译时元编程来分析代码结构、拦截方法及注入行为。

## 16.1 在编译时分析代码

高级开发人员和软件架构师往往会倡导编码标准，而且会尽力确保其团队遵循一致的编程实践。可以利用Groovy的强大将代码审查自动化，以发现代码中的异味[③]和不良实践。本节将介绍如何使用编译时元编程来检查代码异味。

尽管命名变量有些困难，而且需要努力，但是使用只有一个字母的变量名肯定是不对的。与其靠人工监管代码，不如利用编译时元编程来检查差劲的变量名和方法名。

① https://github.com/spockframework

② http://codenarc.sourceforge.net

③ 程序开发领域，代码中的任何可能导致深层次问题的症状都可以叫作代码异味，具体可以参考http://zh.wikipedia.org/wiki/%E4%BB%A3%E7%A0%81%E5%BC%82%E5%91%B3。——译者注

**AST/CodeAnalysis/smelly.groovy**

```
def canVote(a) {
  a > 17 ? "You can vote" : "You can't vote"
}

def p(instance) {
  //用于打印实例的代码
}
```

canVote()方法接受一个表示年龄的参数a，确定这个年龄的人是否可以投票。我们希望自动地探测出代码中存在异味的参数a和古怪的方法名p()。要尽可能早地探测出来，那就在编译代码时。因为这段代码在语法上是正确的，编译器不会检查出其中的异味，但是我们可以。我们可以命令编译器在遇到这种存在异味的代码时不予通过，尽管代码在语法上是正确的。

### 16.1.1 理解代码结构

要检查代码异味，需要遍历代码结构，分析类名、方法名、字段名和参数名等信息。这可以通过编写一个解析器来实现，但是，编译器已经解析并分析过代码，倒不如依靠编译器，尽量减少人为的工作。

Groovy编译器允许我们进入其编译阶段，一窥其所处理的AST（抽象语法树）。AST树结构描述了程序中的表达式和语句，它是使用节点表示的。随着编译过程的进行，程序的AST会被变换，包括节点的插入、删除和重新排列。在编译过程中，我们可以随着AST的演进对它进行检查，加以修改，以及命令编译器去标记警告或错误。

canVote()方法很短，只是返回三元操作符的结果，但是其AST却异常丰富，包含很多细粒度的细节信息（参见图16-1）。

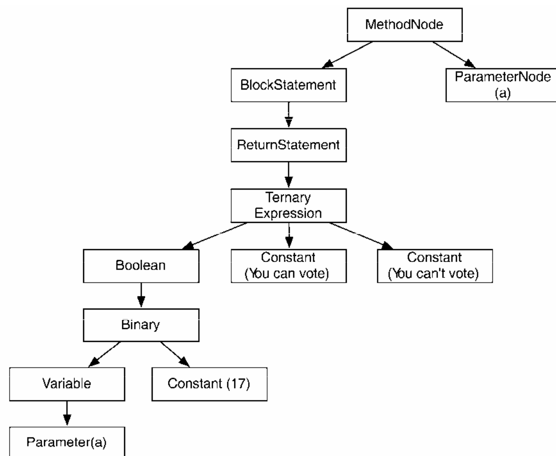



图16-1 canVote()方法的AST

要使用编译时元编程，必须理解和使用AST。这项任务非常复杂，但幸运的是有个帮手。groovyConsole工具有一个很棒的功能，它可以显示代码的AST。在这个工具中打开Groovy源代码，选择Script菜单下的Inspect AST菜单项。groovyConsole工具会显示这段存在异味的代码的AST结构，如图16-2所示。

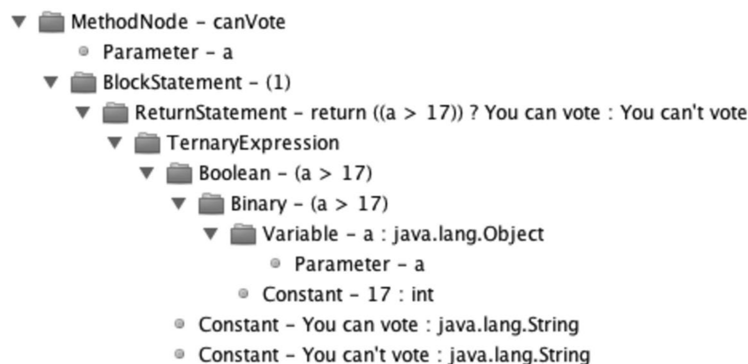



图16-2 在groovyConsole的AST浏览器视图中查看canVote()方法

## 16.1.2 在代码结构中导航

既然已经掌握了可能存在异味的代码的AST，是时候通过在这种结构中导航来检查代码了。Groovy中提供的AST变换应用编程接口（API）使这一任务更易达成了。为了在代码中导航，现在创建一个名为CodeCheck的类，并实现ASTTransformation接口。

**AST/CodeAnalysis/com/agiledeveloper/CodeCheck.groovy**

```
@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class CodeCheck implements ASTTransformation {
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
      sourceUnit.ast.classes.each { classNode ->
        classNode.visitContents(new OurClassVisitor(sourceUnit))
      }
    }
}
```

如果想检查AST，首先使用注解GroovyASTTransformation告知编译器。Groovy编译器包括多个阶段，而且支持开发者在任何阶段中介入：初始化、解析、转换、语义分析、规范化、指令选择、class生成、输出和结束[①]。首个合理的时机是在语义分析阶段之后，AST在此时被创建出来。如果想使用信息更多的AST，可以在更靠后的阶段介入。上面的例子指明该AST变换必须在语义分析阶段（CompilePhase.SEMANTIC_ANALYSIS）之后应用。

① 关于这些编译阶段的更多信息，可以参考http://groovy.codehaus.org/Compiler+Phase+Guide。——译者注

随着编译器到达指定阶段，它将调用用于变换的类的visit()方法，并向该方法提供一个由ASTNode实例组成的列表，以及一个表示被编译代码的SourceUnit引用。可以在visit()方法内迭代这些AST节点来检查各种元素，如果愿意，甚至可以修改这些节点。

在这个例子中，我们想访问整个结构来查找代码异味。AST变换API提供了一个名为GroovyClassVisitor的访问者，简化了我们的操作。对于每个类节点、方法节点、字段节点等，该访问者的方法将被调用，同时被告知相应的节点信息。这时，与其进入类和方法的层次结构，不如坐在原处，在这些方法中采取相应的动作，而将导航这种累活交给API去做。

在visit()方法内，向从给定的源代码单元内找到的每个类节点注册一个GroovyClassVisitor的实现。

现在实现这个GroovyClassVisitor接口。

**AST/CodeAnalysis/com/agiledeveloper/CodeCheck.groovy**

```
class OurClassVisitor implements GroovyClassVisitor {
 SourceUnit sourceUnit
 OurClassVisitor(theSourceUnit) { sourceUnit = theSourceUnit }
 private void reportError(message, lineNumber, columnNumber) {
   sourceUnit.addError(new SyntaxException(message, lineNumber, columnNumber))
 }

 void visitMethod(MethodNode node) {
   if(node.name.size() == 1)
     reportError "Make method name descriptive, avoid single letter names",
         node.lineNumber, node.columnNumber

   node.parameters.each { parameter ->
     if(parameter.name.size() == 1)
       reportError "Single letter parameters are morally wrong!",
           parameter.lineNumber, parameter.columnNumber
   }
 }

 void visitClass(ClassNode node) {}
 void visitConstructor(ConstructorNode node) {}
 void visitField(FieldNode node) {}
 void visitProperty(PropertyNode node) {}
}
```

当导航到一个类中的类元素时，AST变换API将调用访问者的方法；对于构造器、字段、方法等，也是如此。因为现在目的是发现代码异味，所以在visitMethod()方法中检查只有一个字母的方法名，如果找到，则添加一条错误消息。编译器也相应报告该错误，并且让编译失败。也可以不使用错误，而是将其报告为警告。

除了发现方法名中的异味，这里还检查了方法的参数名。可以继续扩展，通过在其他方法中（如visitField()和visitProperty()）实现代码，来发现字段名、属性名等元素中的异味。

这样就几乎实现了一个小型代码异味检查器，但是在编译过程中，还必须帮助编译器发现并应用这个AST变换。这里将使用一种叫作全局变换的方式，此时变换可以应用于任何代码片段中，不需要代码上的任何特殊标记。Groovy编译器将在classpath中的每个JAR文件中查找这样的全局变换。为使查找更为高效，编译器期望我们在各个JAR文件中的一个特殊的清单文件（manifest/META-INF/services/org.codehaus.groovy.transform.ASTTransformation）中声明这个变换的类名。下面就是用于代码检查器这个例子的清单文件的内容：

```
com.agiledeveloper.CodeCheck
```

总结一下这节已经学到的东西。清单文件告知编译器用于变换的类的名字。在变换类中，使用注解指明它应该在哪个编译阶段被调用。在这个例子中，变换类的visit()方法借助一个访问者，调用相应的动作。

现在需要编译CodeCheck类，并使用生成的类文件和清单文件创建一个JAR文件。如果该JAR文件在classpath下，Groovy编译器将应用这个代码异味检查器。下面看一下实现该功能需要的步骤：

```
$ groovyc -d classes com/agiledeveloper/CodeCheck.groovy
$ jar -cf checkcode.jar -C classes com -C manifest .
$ groovyc -classpath checkcode.jar smelly.groovy
```

运行上述命令，检查前面所创建的例子中的代码异味。因为发现了代码异味，编译器将输出一些错误消息，并使编译失败。

```
org.codehaus.groovy.control.MultipleCompilationErrorsException:
  startup failed:
smelly.groovy: 1: Single letter parameters are morally wrong! @
  line 1, column 13.
   def canVote(a) {
               ^

smelly.groovy: 5: Make method name descriptive, avoid single letter names @
  line 5, column 1.
   def p(instance) {
   ^

2 errors
```

代码检查发生在编译阶段，而非运行期间。这需要我们花些时间来熟悉。研究并修改这个例子，随着在AST中导航，设置一些输出信息，仔细了解这种变换如何应用，以及何时应用。

本节介绍了如何使用AST变换来检查代码。如果只是想找出一些常见的代码异味，其实不必这么大费周章。可以使用CodeNarc这款Groovy代码质量工具。它能方便地使用CodeNarc所提供的标准检查方法，甚至可以扩展其规则，以监视想检查的代码异味。现在看的这个例子可以帮我们

见识到AST变换之强大，如果想在代码中执行领域特定的约束检查，AST变换将会派上用场。

AST变换的强大远不止分析代码这么简单。可以在编译时拦截方法调用，甚至向程序中注入代码，下面将予以介绍。

## 16.2　使用 AST 变换拦截方法调用

假设正在开发某款银行软件，业务人员丢给我们一个出乎意料的问题。他们希望，在活期存款账户中，每笔金额超过10 000美元的存款、取款和转账业务都需要经过审计。对此需要做出快速的反应，所以先来考虑几个方案。

最差的方案是找到在所有活期存款账户上执行事务的地方，然后加以修改。但即使凭借我们最喜爱的强大集成开发环境，找到并修改这些调用也不是那么有趣。此外，每次调用其中的一个方法时，还都要记得执行审计操作。

另一个方案是修改活期存款账户类中的方法来执行额外的操作。与第一个方案相比，这个方案更为可靠，而且需要的工作更少。然而，它会导致代码冗余，而且还要当心新加入的函数。

第12章介绍了如何使用运行时元编程拦截方法调用。与前面的两个方案相比，这种技术优势极大。它没有代码冗余，而且对于新加入的方法，这种方案也很容易扩展。但是方法拦截仅发生于运行时，所以会对执行造成轻微影响。本节将介绍如何通过在编译时拦截方法来避免这一不利影响。

看一下包含了新添加的`audit()`方法的`CheckingAccount`类。当这个类中的其他方法被调用时，我们希望该方法也适时调用。

**AST/InterceptingCalls/UsingCheckingAccount.groovy**

```groovy
class CheckingAccount {
  def audit(amount) { if(amount > 10000) print "auditing..." }
  def deposit(amount) { println "depositing ${amount}..." }
  def withdraw(amount) { println "withdrawing ${amount}..." }
}

def account = new CheckingAccount()
account.deposit(1000)
account.deposit(12000)
account.withdraw(11000)
```

现在使用AST变换，在`CheckingAccount`类的每个方法中都加入一个对`audit()`方法的调用（除了该方法本身）。为其编写一个变换类：

**AST/InterceptingCalls/com/agiledeveloper/InjectAudit.groovy**

```groovy
@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class InjectAudit implements ASTTransformation {
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
```

```
    def checkingAccountClassNode =
      astNodes[0].classes.find { it.name == 'CheckingAccount' }
    injectAuditMethod(checkingAccountClassNode)
  }
```

InjectAudit类实现了ASTTransformation接口，并提供了必需的visit()方法。使用@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)注解告诉编译器在语义分析阶段的最后应用该变换。

因为这是一个全局变换，编译器会在被编译的代码中发现的所有节点上触发该变换。在visit()方法内，使用find()方法将代表CheckingAccount类的节点提取出来，之后使用一个辅助方法injectAuditMethod()采取相应的动作：

**AST/InterceptingCalls/com/agiledeveloper/InjectAudit.groovy**

```
static void injectAuditMethod(checkingAccountClassNode) {
  def nonAuditMethods =
    checkingAccountClassNode?.methods.findAll { it.name != 'audit' }
  nonAuditMethods?.each { injectMethodWithAudit(it) }
}
```

使用findAll()方法提取出了除audit()之外的所有方法。之后通过辅助方法injectMethod-WithAudit()，在每个选中的方法开头加入了一个对audit()方法的调用。

真正的动作在injectMethodWithAudit()中。我们希望在每个方法的开头放置一个对audit()方法的调用。遗憾的是，所需的步骤并不是调用一下这个方法那么简单。必须为方法调用创建AST，并将其插入到目标方法AST中的语句列表中。下面实现这一功能：



**AST/InterceptingCalls/com/agiledeveloper/InjectAudit.groovy**

```
  static void injectMethodWithAudit(methodNode) {
    def callToAudit = new ExpressionStatement(
        new MethodCallExpression(
          new VariableExpression('this'),
          'audit',
          new ArgumentListExpression(methodNode.parameters)
        )
    )

    methodNode.code.statements.add(0, callToAudit)
  }
}
```

要理解这里为方法调用创建的AST，可以使用groovyConsole来查看一下该调用对应的内部AST结构。这可以帮助你联想到并弄清楚需要创建什么。像audit(amount)这样的一个简单调用，可能需要很多行代码、一系列表达式对象，如上面的代码所示。

MethodCallExpression在AST层次表示这个调用。其第一个参数——VariableExpression，

指出这个调用要在此执行环境中的当前对象（this）上进行。第二个参数指出了要调用的方法的名字。第三个参数代表要传给这个方法的参数，在这个例子中就是一个使用外围方法节点的参数创建的ArgumentListExpression实例。

最后，将所创建的表达式节点插入到目标方法的语句列表中。

在将该变换投入使用之前还有最后一步，即必须让编译器知道它。创建清单文件META-INF/services/org.codehaus.groovy.transform.ASTTransformation，在其中列出该变换的类名。

```
com.agiledeveloper.InjectAudit
```

首先编译该变换，并将其打入JAR包。

```
$ groovyc -d classes com/agiledeveloper/InjectAudit.groovy
$ jar -cf injectAudit.jar -C classes com -C manifest .
```

用于将方法调用加入到CheckingAccount类中的变换就准备好了。为研究这一变换的作用，首先不使用该变换来运行UsingCheckingAccount.groovy：

```
$ groovy UsingCheckingAccount.groovy
```

调用3个方法的结果就是直接调用：

```
depositing 1000...
depositing 12000...
withdrawing 11000...
```

之后再使用这一变换来修改其行为，将injectAudit.jar包含在classpath中。

```
$ groovy -classpath injectAudit.jar UsingCheckingAccount.groovy
```

编译器将识别出injectAudit.jar中的这一变换，并加入对audit()方法的调用：

```
depositing 1000...
auditing...depositing 12000...
auditing...withdrawing 11000...
```

每次调用CheckingAccount类上的一个方法，都会先调用audit()，但是这一变换不会影对audit()方法的任何直接调用。

现在是以脚本形式运行的代码，要使变换生效，每次运行时都要将injectAudit.jar包含在classpath中。为避免该问题，可以预编译这段代码。只需要使用groovyc编译这个脚本，并将injectAudit.jar放到classpath中。生成的字节码中将包含对audit()的适当调用。之后就可以使用groovy或java命令运行字节码了。

本节使用AST变换在编译时添加了方法调用。与运行时元编程相比，性能更好，然而要做的工作多出很多。下面将介绍缓解这一问题的不同方法。

像this.audit()这样一个简单的调用，在变换期间都需要创建多个对象、很多行代码。对于更复杂的调用，想想就令人生畏，甚至最有激情的程序员都会很快望而却步。谢天谢地，

ASTBuilder类可以减轻我们的负担。

ASTBuilder提供了3种创建AST子树的不同方式：buildFromSpec()、buildFromString()和buildFromCode()。下面使用它们实现injectMethodWithAudit()方法。

实例化表达式等类的实例这种新操作有很多噪音，buildFromSpec()方法可以帮助减少这些噪音。简单地使用一个methodCall块来创建一个MethodCallExpression实例，并使用variable来定义一个变量，如下面这个版本的injectMethodWithAudit()所示：

**AST/EasingThePain/com/agiledeveloper/InjectAudit.groovy**

```
    static void injectMethodWithAudit(methodNode) {
      List<Statement> statements = new AstBuilder().buildFromSpec {
        expression {
          methodCall {
            variable 'this'
            constant 'audit'
            argumentList {
              methodNode.parameters.each { variable it.name }
            }
          }
        }
      }
      def callToCheck = statements[0]
      methodNode.code.statements.add(0, callToCheck)
    }
}
```



使用buildFromSpec()方法创建AST非常流畅，但是这种方式也引入了一些复杂性——必须熟悉该API所期望的DSL语法。为此我们必须知道正在创建的AST的结构，毕竟该API只是使语法更流畅了。

与其花费这么大力气，不如简单地使用buildFromString()方法，从嵌入在字符串中的一段代码获得一个AST变换。下面使用这一生成器方法重写injectMethodWithAudit()方法。

**AST/EasingThePain2/com/agiledeveloper/InjectAudit.groovy**

```
    static void injectMethodWithAudit(methodNode) {
      def codeAsString = 'audit(amount)'
      List<Statement> statements = new AstBuilder().buildFromString(codeAsString)

      def callToAudit = statements[0].statements[0].expression
      methodNode.code.statements.add(0, new ExpressionStatement(callToAudit))
   }
}
```

简单地将想插入的语句丢到了一个字符串中，剩下的活交给生成器处理。在创建AST时，buildFromString()给我们省了很多力气，但是遗憾的是，生成的结果被包在了一个return语

句中。必须再花点力气将所需的内容提取出来，在将其插入到目标方法的语句中之前，需要先将其放到一个ExpressionStatement中。

buildFromString()方法还有一个问题，它要求我们将代码放到一个字符串中。处理转义字符和多行代码会很麻烦。就算借助here文档语法（参见5.3节），也是非常困难。

buildFromCode()优于其他方法。可以像自然编写代码那样编写代码，并将其放到一个代码块中。buildFromCode()就像一位善良的撒玛利亚人[1]，接收这个代码块，并生成AST变换。下面使用这个方法重写injectMethodWithAudit()方法：

**AST/EasingThePain3/com/agiledeveloper/InjectAudit.groovy**

```
  static void injectMethodWithAudit(methodNode) {
    List<Statement> statements = new AstBuilder().buildFromCode { audit(amount) }
    def callToAudit = statements[0].statements[0].expression
    methodNode.code.statements.add(0, new ExpressionStatement(callToAudit))
  }
}
```

这种方式对我们帮助很大。

- 不必再为要创建的节点的AST结构而挣扎。
- 不必担心AST结构是否会在未来的Groovy版本中发生改变；AstBuilder会随之演化，将我们从修改AST结构的工作中解放出来。
- 显而易见，所要生成的代码没有迷失在AST结构的细节之中。

buildFromCode()方法非常吸引人，但在使用时有一些注意事项：它没有完全解放我们，我们仍然需要对AST结构有所了解；仍然必须从产生的AST中提取正确的部分，并需要知道将其置于何处。对于可以使用该方法生成的代码，也存在一些限制：生成的代码会放在该变换编译后的代码中，躲不开窥视的眼睛。最后，ASTBuilder的构建方法本身也要经过一次编译时的AST变换，这限制我们只能使用Groovy编写变换代码，而没有使用ASTBuilder的变换则可以使用任何JVM语言编写。

本节介绍了如何在编译时拦截方法并向其中添加行为。也可以使用该技术向类中添加新的方法和字段，下一节将予以介绍。

## 16.3　使用 AST 变换注入方法

上一节介绍了如何向已有的方法中注入代码。通过使用AST变换，也可以向类中注入方法和字段。这样无需第三方库，即可在编译时发挥AOP的全部威力。

① Samaritan，中东种族撒马利亚人，因心肠好而著称。——译者注

4.5节介绍的Execute Around Method模式中，使用了一个静态方法来辅助创建和清理实例。该方法支持在两个操作之间随便使用它生成的实例。下面使用AST变换来实现该模式。与其让程序员手动实现use()方法，不如我们来创建它，程序员只要请求调用即可。漂亮！

到目前为止，本章介绍的AST变换都是全局变换。它们会被应用于被编译的所有代码上。在变换内确定是要执行某些变换，还是简单地跳过。那种方式是非侵入式的。被变换的代码无须关注该变换，也不需要任何特殊处理。

然而对于有些可能遇到的问题，那种方式太过全面。这里需要知道把use()方法注入到哪个类中。本节的亮点就在于此。下面编写一个局部变换，该变换只应用于程序员使用所提供的特殊注解标记的选定部分。局部变换有一个优点：不必创建额外的清单文件。

先创建会触发变换的注解。

**AST/EAM/com/agiledeveloper/EAM.groovy**

```
@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.TYPE])
@GroovyASTTransformationClass("com.agiledeveloper.EAMTransformation")

public @interface EAM {
}
```

使用Target，指明这个注解只能放在类上。使用GroovyASTTransformationClass，告知编译器，参数中提到的变换com.agiledeveloper.EAMTransformation应该应用于使用这个EAM注解标记的任何类。



下面将静态的use()方法插入到被注解的类中。这听上去有点吓人，但是编写局部变换与编写全局变换并没有多大区别，而后者我们已经会了。因为变换只在目标类上调用，所以可以直接在visit()方法中处理：

**AST/EAM/com/agiledeveloper/EAMTransformation.groovy**

```
Line 1 @GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
     - class EAMTransformation implements ASTTransformation {
     -     void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
     -
     5       astNodes.findAll { node -> node instanceof ClassNode }.each { classNode ->
     -
     -         def useMethodBody = new AstBuilder().buildFromCode {
     -           def instance = newInstance()
     -           try {
    10             instance.open()
     -             instance.with block
     -           } finally  {
     -             instance.close()
     -           }
```

```
15          }
-
-           def useMethod = new MethodNode(
-             'use', ACC_PUBLIC | ACC_STATIC, ClassHelper.OBJECT_TYPE,
-             [new Parameter(ClassHelper.OBJECT_TYPE, 'block')] as Parameter[],
20            [] as ClassNode[], useMethodBody[0])
-
-           classNode.addMethod(useMethod)
-         }
-       }
25  }
```

EAM模式的核心是我们想注入到类中的特殊的use()方法。在这个方法中，需要将实例发送给一个使用该实例的闭包。闭包调用本身需要包在一个try-finally块中，finally块中将调用清理代码。

在visit()方法内，使用ASTBuilder的buildFromCode()方法来创建use()方法，并将其注入到类中。这里假设目标类有一个open()方法和一个close()方法。如果缺少这两个方法，则将抛出运行时错误。如果愿意，也可以抛出编译时错误。为此，必须遍历该类的AST节点，如果没有找到这些方法，则像16.1节所做的那样报告错误。

在visit()方法中，第7行使用buildFromCode()创建的只是use()方法的方法体。之后还必须将方法体附到一个表示use()方法的方法节点上。第17行创建了一个MethodNode实例，并将方法体附了上去。

下面仔细看一下MethodNode的创建。第一个参数指明了方法名（use）。第二个参数指明了该方法应该使用public和static修饰符。第三个参数指明该方法的返回类型（Object）。方法一般都要接收参数，但是这里的use()方法期待的是一个闭包。代码中使用第四个参数指明了这一点，该参数是一个列表，需要列出use()方法所需的每个参数的类型和名字。第五个参数指明方法可能会抛出的异常，这个例子中没有。最后一个参数指向的是所创建方法的方法体。

最后一步，使用addMethod()方法将刚创建的这个方法添加到目标类中。得到这段简洁的代码真是费了不少劲，现在已经设计出一种方式，实现了向使用EAM注解标记的任何类中注入use()方法。下面找个类试试，不过首先需要编译变换代码，并将其打包到一个JAR文件中。

```
$ groovyc -d classes \
  com/agiledeveloper/EAM.groovy \
  com/agiledeveloper/EAMTransformation.groovy
$ jar -cf eam.jar -C classes com
```

创建一个可以注入use()方法的Resource类。

**AST/EAM/resource.groovy**

```
@com.agiledeveloper.EAM
class Resource {
```

```
  private def open() { print "opened..." }
  private def close() { print "closed" }
  def read() { print "read..." }
  def write() { print "write..." }
}
println "Using Resource"
Resource.use  {
  read()
  write()
}
```

Resource类提供了期望的open()和close()方法。代码中调用了期望的use()方法。不要担心这个方法不存在，因为使用EAM注解标记了Resource 类，前面编写的变换会向这个类中注入use()方法。最后，当编译Resource类时，要确保eam.jar在classpath下：

```
$ groovy -classpath eam.jar resource.groovy
```

从该命令的输出可以看出，通过编译时元编程实现的EAM模式起作用了。

```
Using Resource
opened...read...write...closed
```

4.5节创建了use()方法，而这一节通过AST变换，将该方法注入到了Resource类或任何使用@EAM注解的类中。一旦驾驭了AST变换，就可以使用这种技术实现非常强大的变换。

警告：在使用如此强大的工具时，需要确保变换的表现确实符合预期。好在这方面我们也有一些帮手。Groovy提供了一个可以使用@ASTTest注解调用的AST变换，以帮助测试其他AST变换，以及在不同的AST节点上对预期结果施加断言。[①]

16

元编程是最强大的概念之一。如果使用得当，它可以帮助创建高度可扩展的软件。像Grails这样的框架就大量使用了元编程。Groovy的特殊之处在于，它同时提供了运行时和编译时的元编程能力。本章探讨的内容不只是如何借助这种能力使用Groovy语言，还包括如何灵活地使用这种能力向现有代码中注入行为。

本书的第三部分介绍了如何立即创建类、方法和属性。可以拦截对已有的方法的调用，甚至还可以拦截对不存在的方法的调用。使用元编程的程度取决于应用特定的需求。不过我们知道，当应用需要元编程时，可以快速实现。第四部分将介绍一些元编程可以起到关键作用的场景，比如使用模拟对象进行单元测试、创建生成器和创建DSL等。

① 参见http://groovy.codehaus.org/gapi/groovy/transform/ASTTest.html。

*Part 4*

第四部分

# 使用元编程

本部分内容

第 17 章

# Groovy生成器

17

生成器是内部的DSL，为处理某些特定类型的问题提供了方便。举个例子，如果需要使用嵌入的、层次式结构，比如树结构、XML、HTML或JSON（JavaScript Object Notation）等表示形式，生成器会非常有用。生成器提供的语法不会把使用者紧紧地绑定到底层的结构或实现上。生成器也不会替换掉底层实现；相反，它们只是为处理底层实现提供了一种优雅的方式。

Groovy可以用于很多日常任务，包括处理XML、JSON、HTML、DOM、SAX、Swing甚至Ant。在本章中，我们将通过一些任务来感知生成器，之后再研究2种创建生成器的技术。

## 17.1　构建 XML

程序员中的大部分人都讨厌XML。随着文档的增大，处理XML会越来越困难， 工具和API支持也不尽如人意。我的理论是，XML就像人： 小时候聪明可爱，越大越招人烦。

XML可能是一种很适合机器处理的格式，但是人直接处理很不方便。没有人会真喜欢处理XML，但是又不得不做。而Groovy几乎使得处理XML变成了一种乐趣，极大地缓解了其中的痛苦。

下面看一个例子，在Groovy中使用生成器创建XML文档：

**UsingBuilders/UsingXMLBuilder.groovy**

```
bldr = new groovy.xml.MarkupBuilder()
bldr.languages {
  language(name: 'C++') { author('Stroustrup')}
  language(name: 'Java') { author('Gosling')}
  language(name: 'Lisp') { author('McCarthy')}
}
```

这段代码使用`groovy.xml.MarkupBuilder`来创建XML文档。当在生成器上调用任意的方法或属性时，它会根据调用的上下文，体贴地假定我们引用的是所生成文档中的元素名或属性名。上述代码的输出如下：

```
<languages>
  <language name='C++'>
    <author>Stroustrup</author>
  </language>
  <language name='Java'>
    <author>Gosling</author>
  </language>
  <language name='Lisp'>
    <author>McCarthy</author>
  </language>
</languages>
```

我们调用了一个名为`languages()`的方法，但该方法在`MarkupBuilder`类的实例上并不存在。不过生成器并没有拒绝它，而是聪明地假定这次调用其实是想定义XML文档的一个根元素，这种假定可真不错。

附在这个方法调用上的闭包现在提供了一个内部的上下文。领域特定语言是与上下文有关的。在这个闭包内，被调用到的任何不存在的方法，都会被假定为是一个子元素的名字。如果在调用方法时传递的是Map参数（如`language(name: value)`），它们会被当作元素的属性。任何单个的参数值（如`author(value)`），表示的是元素内容，而非属性。可以研究一下前面的代码和相关输出，看看`MarkupBuilder`是如何推断代码的。

在前面的示例中，进入XML文档的数据都是硬编码的，而且生成器只是将结果写到了标准输出中。而在实际的项目中，这两种情况都很少用到。我们的数据可能来自一个集合，而集合又可能是通过一个数据源或输入流填充的。此外，我们可能还想把XML内容写入到一个`Writer`中，而不是写入到标准输出中。

生成器上可以轻松地附上一个`Writer`，将其作为构造器的一个参数。所以，可以将一个`StringWriter`附到生成器上。数据可以来自任何源，比如来自数据库（参见9.3节）。下面例子从一个`Map`中取到数据，创建了一个XML文档，并将文档写到了一个`StringWriter`中：



**UsingBuilders/BuildXML.groovy**

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

writer = new StringWriter()
bldr = new groovy.xml.MarkupBuilder(writer)
bldr.languages {
  langs.each { key, value ->
      language(name: key) {
        author (value)
      }
    }
}
println writer
```

这段代码的输出如下：

```
<languages>
  <language name='C++'>
    <author>Stroustrup</author>
  </language>
  <language name='Java'>
    <author>Gosling</author>
  </language>
  <language name='Lisp'>
    <author>McCarthy</author>
  </language>
</languages>
```

MarkupBuilder十分适合小到中型的文档。然而，如果文档非常大（若干兆字节），我们可以使用StreamingMarkupBuilder，它的内存占用情况更好一些。下面使用StreamingMarkupBuilder重写前面的示例，为增加一些趣味性，我们把命名空间和XML注释包含了进来：

**UsingBuilders/BuildUsingStreamingBuilder.groovy**

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

xmlDocument = new groovy.xml.StreamingMarkupBuilder().bind {
  mkp.xmlDeclaration()
  mkp.declareNamespace(computer: "Computer")
  languages {
    comment << "Created using StreamingMarkupBuilder"
    langs.each { key, value ->
      computer.language(name: key) {
        author (value)
      }
    }
  }
}
println xmlDocument
```

新版本的代码产生的输出如下：

```
<?xml version="1.0"?>
<languages xmlns:computer='Computer'>
  <!--Created using StreamingMarkupBuilder-->
    <computer:language name='C++'>
      <author>Stroustrup</author>
    </computer:language>
    <computer:language name='Java'>
      <author>Gosling</author>
    </computer:language>
    <computer:language name='Lisp'>
      <author>McCarthy</author>
    </computer:language>
</languages>
```

利用StreamingMarkupBuilder，借助该生成器支持的属性mkp，可以声明命名空间、XML注释等内容。一旦定义了一个命名空间，要将元素与命名空间关联起来，在前缀上使用点记号(.)即可，如computer.language，这里computer就是一个前缀。

XML的生成器，语法简单且优雅。我们不必使用XML的复杂语法来创建XML文档。创建XML输出也非常容易。然而如果要创建的是JSON输出，Groovy也考虑到了，下一节将会介绍。

## 17.2 构建 JSON

当创建Web服务，需要生成JSON格式的对象时，Groovy也提供了方便的解决方案[1]。只需将实例发送给groovy.json.JsonBuilder的构造器，这个生成器会处理余下的工作，就这么简单。通过调用writeTo()方法，可以将生成的JSON格式写入到一个Writer中，如下面的例子所示：

**UsingBuilders/BuildJSON.groovy**

```
class Person {
  String first
  String last
  def sigs
  def tools
}
john = new Person(first: "John", last: "Smith",
  sigs: ['Java', 'Groovy'], tools: ['script': 'Groovy', 'test': 'Spock'])
bldr = new groovy.json.JsonBuilder(john)
writer = new StringWriter()
bldr.writeTo(writer)
println writer
```

该生成器使用字段的名字以及它们的值作为JSON格式的键和值，如下所示：

```
{"first":"John","last":"Smith","tools":{"script":"Groovy","test":"Spock"},
  "sigs":["Java","Groovy"]}
```



产生输出毫不费力。如果想对输出加以定制，也只需要多加几步。利用生成器流畅地创建指定的输出，如下面的例子所示。

**UsingBuilders/BuildJSON.groovy**

```
bldr = new groovy.json.JsonBuilder()
bldr {
  firstName john.first
  lastName john.last
  "special interest groups" john.sigs
  "preferred tools" {
    numberOfTools john.tools.size()
```

① http://www.json.org/

```
    tools john.tools
  }
}
writer = new StringWriter()
bldr.writeTo(writer)
println writer
```

这里没有直接使用实例的属性名，而是为每个属性选择了不同的名字。还可以添加新属性，比如这个例子中的numberOfTools。生成器会使用我们提供的DSL语法来创建输出，其内容如下：

```
{"firstName":"John","lastName":"Smith",
  "special interest groups":["Java","Groovy"],
  "preferred tools":{"numberOfTools":2,
    "tools":{"script":"Groovy","test":"Spock"}}}
```

JsonBuilder可以从JavaBean、HashMap和列表生成JSON格式的输出。JSON格式的输出被保存在内存中，可以稍后再将其写入到一个流中，或是将其用于进一步处理。如果不想将数据保存在内存中，而想在创建时就直接将其变为流，可以使用StreamingJsonBuilder代替JsonBuilder。

在Groovy中，反方向的处理也很容易；比如我们可以利用Groovy提供的JsonSlurper，从JSON数据创建HashMap。可以使用parseText()方法读取包含在String中的JSON数据。也可以使用parse() 方法从Reader或文件中读取JSON数据。

下面我们解析一下前面例子中创建的JSON输出，假设现在输出保存在person.json文件中。

**UsingBuilders/person.json**

```
{"first":"John","last":"Smith","tools":{"script":"Groovy","test":"Spock"},
  "sigs":["Java","Groovy"]}
```

除了来自文件，JSON数据也有可能来自一个Web服务。一旦以一个Reader实例的形式获得了数据流，就可以像下面例子中这样将其传给parse()方法。下面是处理person.json文件中的JSON数据的代码：

**UsingBuilders/ParseJSON.groovy**

```
def sluper = new JsonSlurper()
def person = sluper.parse(new FileReader('person.json'))

println "$person.first $person.last is interested in ${person.sigs.join(', ')}"
```

我们创建了一个FileReader，去读取文件中的数据。之后将其传给parse()方法，该方法会返回一个包含数据的HashMap实例。既可以像这里这样使用HashMap中的键和值，也可以从这些数据创建一个 Groovy对象——还记得吗，可以以HashMap作为构造器的参数创建Groovy对象（参见2.2节）。

解析JSON数据的代码的输出如下：

```
John Smith is interested in Java, Groovy
```

解析JSON数据不费什么劲，简单得让人不安正是其便利性所在。

Groovy的生成器不仅仅能够生成数据，它们甚至可以让Swing编程体验变得相当漂亮，下一节将会介绍。

## 17.3 构建 Swing 应用

生成器之优雅并不局限于XML结构，Groovy还为创建Swing应用提供了一个生成器。当使用Swing时，我们需要执行一些很乏味的任务，比如创建组件（如按钮）、注册事件处理器等。通常，要实现一个事件处理器，要编写一个匿名内部类，而在实现处理器方法时，有些参数即使我们并不关心（如`ActionEvent`），也得接受它们。`SwingBuilder`，结合Groovy的闭包，帮我们去掉了这种苦差事。

可以使用生成器提供的嵌套或层次化结构来创建一个容器（如`JFrame`）及其组件（如按钮、文本框等）。Groovy提供的灵活的键值对可以初始化设施来初始化组件。定义事件处理器是小菜一碟，只需要向生成器提供一个闭包。尽管正在构建的Swing应用并不陌生，但是我们会发现，与用Java编写相比，用Groovy编写需要的代码要少一些。这有助于快速修改、试验并获得反馈。尽管仍在使用底层的Swing API，但语法有很大的不同。我们是在使用Groovy的方言与Swing对话[①]。现在使用`SwingBuilder`类来创建一个Swing应用：

**UsingBuilders/BuildSwing.groovy**

```
bldr = new groovy.swing.SwingBuilder()

frame = bldr.frame(
  title: 'Swing',
  size: [50, 100],
  layout: new java.awt.FlowLayout(),
  defaultCloseOperation:javax.swing.WindowConstants.EXIT_ON_CLOSE
) {
  lbl = label(text: 'test')
  btn = button(text: 'Click me', actionPerformed: {
    btn.text = 'Clicked'
    lbl.text = "Groovy!"
    } )
}

frame.show()
```

图17-1显示了前面代码的输出。

① http://blog.agiledeveloper.com/2007/05/its-not-languages-but-their-idioms-that.html

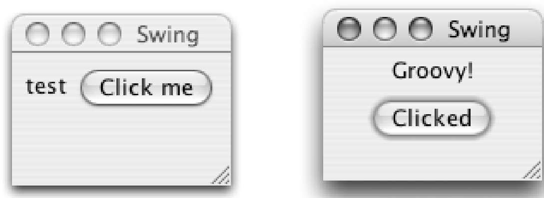



图17-1　使用SwingBuilder创建的一个小型Swing应用

这里初始化了一个JFrame实例，指定了它的title（标题）、size（大小）和layout（布局），并设置了默认的关闭操作，一个简单的语句就搞定了这一切。这等价于Java中5条单独的语句。此外，注册事件处理器也很简单，只需要向button（这里代表JButton）的actionPerformed属性提供一个闭包。在Java中，要创建一个匿名内部类，并使用ActionEvent参数实现actionPerformed()方法，在Groovy中则不必这么辛苦。当然，这里有很多语法糖，但是看上去非常优雅，而且减少了代码量，这使得Swing API更易用了。

SwingBuilder生成器向我们演示了Groovy强大的表现力。它非常迷人，但是如果要创建任何比较大型的Swing应用，建议研究一下Griffon项目[①]。Griffon是构建于Groovy之上的一个框架，用于使用“约定优于配置”（Convention Over Configuration）的原则创建Swing应用。它不仅减轻了创建GUI的痛苦，还可以跨多个线程正确地处理事件。

## 17.4　使用元编程定制生成器

前面介绍过，有一些使用了嵌套或层次化结构或格式的专门化而且非常复杂的任务，对于这种任务，生成器提供了一种创建内部DSL的方式。当在应用中处理专门化的任务时，可以检查一下，是不是有生成器可以解决这个问题。如果没有找到任何生成器，可以自行创建。

定制生成器有两种创建方式：利用Groovy的元编程能力，一切自己来，本节采用的就是这种方式；使用Groovy提供的BuilderSupport（参见17.5节）或FactoryBuilderSupport（参见17.6节）。

为帮助理解BuilderSupport的优势，下面构建一个待办事项列表。下面的代码中使用了本节即将创建的生成器：

**UsingBuilders/UsingTodoBuilder.groovy**

```
bldr = new TodoBuilder()

bldr.build {
  Prepare_Vacation (start: '02/15', end: '02/22') {
    Reserve_Flight (on: '01/01', status: 'done')
```

① http://griffon.codehaus.org.

```
    Reserve_Hotel(on: '01/02')
    Reserve_Car(on: '01/02')
  }
  Buy_New_Mac {
    Install_QuickSilver
    Install_TextMate
    Install_Groovy {
      Run_all_tests
    }
  }
}
```

在创建完ToDoBuilder之后，将会看到这段代码的输出如下：

```
To-Do:
 - Prepare Vacation [start: 02/15 end: 02/22]
  x Reserve Flight [on: 01/01]
  - Reserve Hotel [on: 01/02]
  - Reserve Car [on: 01/02]
 - Buy New Mac
  - Install QuickSilver
- Install TextMate
- Install Groovy
 - Run all tests
```

完成的任务使用x标记。缩进说明了任务的嵌套层次，而诸如开始日期等任务参数会紧跟在任务名称后面。

在上述用于待办事项列表的DSL中，我们创建了以诸如Reserve Car为名称的条目，不过这里用下划线代替了空格，这样就可以将其用作Groovy中的方法名了。build()是其中唯一一个提前确定的方法。其余的方法和属性都是通过methodMissing()和propertyMissing()来处理的，如下所示：



UsingBuilders/TodoBuilder.groovy

```
class TodoBuilder {
  def level = 0
  def result = new StringWriter()
  def build(closure) {
    result << "To-Do:\n"
    closure.delegate = this
    closure()
    println result
  }

  def methodMissing(String name, args) {
    handle(name, args)
  }

  def propertyMissing(String name) {
```

```
    Object[] emptyArray = []
    handle(name, emptyArray)
  }

  def handle(String name, args) {
    level++
    level.times { result << " "}
    result << placeXifStatusDone(args)
    result << name.replaceAll("_", " ")
    result << printParameters(args)
    result << "\n"

    if (args.length > 0 && args[-1] instanceof Closure) {
      def theClosure = args[-1]
      theClosure.delegate = this
      theClosure()
    }

    level--
  }
  def placeXifStatusDone(args) {
    args.length > 0 && args[0] instanceof Map &&
      args[0]['status'] == 'done' ? "x " : "- "
  }

  def printParameters(args) {
    def values = ""
    if (args.length > 0 && args[0] instanceof Map) {
      values += " ["
      def count = 0
      args[0].each { key, value ->
        if (key == 'status') return
        count++
        values += (count > 1 ? " " : "")
        values += "${key}: ${value}"
      }
      values += "]"
    }

    values
  }
}
```

几乎全部是标准直接的Groovy代码，而且很好地应用了元编程。当被调用到不存在的方法或属性时，就假定它是一个条目。为检查调用时是不是提供了闭包，这里以-1为索引，获得了args中的最后一个参数，并对它进行了测试。之后将当前闭包的delegate设置为生成器，并调用该闭包向下遍历嵌套的任务。

创建定制生成器并不困难，不要犹豫。对于嵌套层次较深，而且大量使用Map和普通参数的

非常复杂的情况，下一节要介绍的BuilderSupport会有所帮助。

## 17.5 使用 BuilderSupport

上一节介绍了如何使用methodMissing()和propertyMissing()创建定制的生成器。如果要创建的生成器不止一个，可能要将某些方法识别代码重构到一个公共基类中。好在Groovy已经这么做了，BuilderSupport类提供了用于识别节点结构的便捷方法。这样就不用亲自编写处理结构的逻辑了，而只需要简单地监听调用，因为Groovy会遍历结构并采取相应的动作。扩展抽象类BuilderSupport感觉就像使用SAX（Simple API for XML，一个流行的事件驱动的XML解析器）。在解析与识别文档中的元素和属性时，它会触发我们提供的处理器上的事件。

在探索如何实现生成器之前，先来看看它能干什么：

**UsingBuilders/UsingTodoBuilderWithSupport.groovy**

```
bldr = new TodoBuilderWithSupport()

bldr.build {
  Prepare_Vacation (start: '02/15', end: '02/22') {
    Reserve_Flight (on: '01/01', status: 'done')
    Reserve_Hotel(on: '01/02')
    Reserve_Car(on: '01/02')
  }
  Buy_New_Mac {
    Install_QuickSilver
    Install_TextMate
    Install_Groovy {
      Run_all_tests
    }
  }
}
```

在创建完毕ToDo-BuilderWithSupport后再来运行前面的代码，输出如下：

```
To-Do:
 - Prepare Vacation [start: 02/15 end: 02/22]
  x Reserve Flight [on: 01/01]
  - Reserve Hotel [on: 01/02]
  - Reserve Car [on: 01/02]
 - Buy New Mac
  - Install QuickSilver
  - Install TextMate
  - Install Groovy
   - Run all tests
```

BuilderSupport期望我们实现两组具体的方法：setParent()和重载版本的createNode()。也可以视情况实现其他方法，比如nodeCompleted()。请记住在调用方法时可做的选

择：可以不提供参数（foo()），可以提供某个值（foo(6)），也可以提供一个Map（foo(name: 'Brad', age: 12)），还可以提供一个Map和一个值（foo(name:'Brad', age:12, 6)）。BuilderSupport提供了4个版本的createNode()，分别对应这4种选择。当在生成器实例上调用方法时，相应的方法会被调用。setParent()用于让生成器的作者知道当前所处理节点的父节点。不管createNode()返回的是什么，都会被看作一个节点，而且生成器支持将其作为一个参数发送给nodeCompleted()。

BuilderSupport不会像处理缺失的方法那样处理缺失的属性。然而仍然可以使用propertyMissing()方法来处理那些情况。

下面来看一下TodoBuilderWithSupport的代码，它扩展了BuilderSupport。待办事项列表所选择的格式仅支持无参数的方法调用（和属性）以及接受一个Map的方法调用。所以接受一个Object类型参数的createNode()版本会抛出一个异常，指示这是一种无效格式。在该方法的另外两个版本，以及propertyMissing()方法中，我们会通过增加level变量记录嵌套层次。在nodeCompleted()方法中要减少level，因为当退出一个嵌套层次时才会调用这个方法。createNode()方法返回所创建节点的名字， 所以我们可以将这个名字与nodeCompleted()比较，以确定何时退出最顶层的节点build。如果需求更为复杂，可以亲自定制表示不同节点的类，并返回这个类的实例。当节点被创建时，如果需要执行一些其他操作，比如将子节点附到父节点上，可以使用setParent()来实现。该方法接受的参数是用作父节点和子节点的Node实例，也就是创建这些节点时createNode()所返回的节点对象。TodoBuilderWithSupport中的其余代码用于处理所发现的节点并创建期望的输出。

尝试一下，看看哪些方法被调用了，又是以什么样的顺序调用的。为理解其顺序，可以在这些方法中插入一些println语句。

**UsingBuilders/TodoBuilderWithSupport.groovy**

```groovy
class TodoBuilderWithSupport extends BuilderSupport {
  int level = 0
  def result = new StringWriter()
  void setParent(parent, child) {}

  def createNode(name) {
    if (name == 'build') {
      result << "To-Do:\n"
      'buildnode'
    } else {
      handle(name, [:])
    }
  }
  def createNode(name, Object value) {
    throw new Exception("Invalid format")
  }
```

```
  def createNode(name, Map attribute) {
    handle(name, attribute)
  }
  def createNode(name, Map attribute, Object value) {
    throw new Exception("Invalid format")
  }

  def propertyMissing(String name) {
    handle(name, [:])
    level--
  }

  void nodeCompleted(parent, node) {
    level--
    if (node == 'buildnode') {
      println result
    }
  }

  def handle(String name, attributes) {
    level++
    level.times { result << " "}
    result << placeXifStatusDone(attributes)
    result << name.replaceAll("_", " ")
    result << printParameters(attributes)
    result << "\n"
    name
  }

  def placeXifStatusDone(attributes) {
     attributes['status'] == 'done' ? "x " : "- "
  }

  def printParameters(attributes) {
    def values = ""
    if(attributes.size() > 0) {
      values += " ["
      def count = 0
      attributes.each { key, value ->
        if (key == 'status') return
        count++
        values += (count > 1 ? " " : "")
        values += "${key}: ${value}"
      }
      values += "]"
    }
    values
  }
}
```

我们看到了将公用代码重构到BuilderSupport中的优势，但是还可以利用另一个层次的重构，下一节将会介绍。

## 17.6　使用 FactoryBuilderSupport

如果要处理的是SwingBuilder中的button（按钮）、checkbox（复选框）和label（标签）等明确定义的节点名，将用到FactoryBuilderSupport。上一节介绍的BuilderSupport适合处理层次式结构，然而对于处理不同类型的节点并不方便。假设必须处理20种不同类型的节点，createNode()的实现就会变得非常复杂。基于节点名创建不同的节点，会导致出现大量麻烦的switch语句。我们可能很快就倾向于使用抽象工厂（参见*Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]）方式来创建这些节点。FactoryBuilderSupport就是这么做的。基于节点名，将节点的创建委托给不同的工厂。我们只需要将节点名映射到工厂。

FactoryBuilderSupport受到了SwingBuilder的启发，后来SwingBuilder又做了修改，扩展了FactoryBuilderSupport，而不再扩展BuilderSupport。下面看一个实现和使用扩展了FactoryBuilderSupport的生成器的例子。创建一个名为RobotBuilder的生成器，可以将其用于机器人的创建与编程。作为第一步，先考虑一下怎么使用这个生成器：

**UsingBuilders/UsingFactoryBuilderSupport.groovy**

```
def bldr = new RobotBuilder()

def robot = bldr.robot('iRobot') {
    forward(dist: 20)
    left(rotation: 90)
    forward(speed: 10, duration: 5)
}

robot.go()
```

我们希望RobotBuilder从代码中产生如下输出：

```
Robot iRobot operating...
move distance... 20
turn left... 90 degrees
move distance... 50
```

现在看一下这个生成器。RobotBuilder扩展了FactoryBuilderSupport。在它的实例初始化器中，使用了FactoryBuilderSupport的registerFactory()方法，将节点名robot、forward和left映射到了相应的工厂。RobotBuilder中就这么多东西。像遍历节点层次、调用相应工厂这些苦差事，FactoryBuilderSupport都给干了。下面将会看到，其余的细节工作，将由工厂和节点负责：

**UsingBuilders/UsingFactoryBuilderSupport.groovy**

```
class RobotBuilder extends FactoryBuilderSupport {
  {
```

```
    registerFactory('robot', new RobotFactory())

    registerFactory('forward', new ForwardMoveFactory())

    registerFactory('left', new LeftTurnFactory())
  };
}
```

下面显示的Robot、ForwardMove和LeftTurn等类分别表示robot、forward和left节点。

**UsingBuilders/UsingFactoryBuilderSupport.groovy**

```
class Robot {
  String name
  def movements = []

  void go() {
    println "Robot $name operating..."
    movements.each { movement -> println movement }
  }
}

class ForwardMove {
  def dist
  String toString() { "move distance... $dist"}
}

class LeftTurn {
  def rotation
  String toString() { "turn left... $rotation degrees"}
}
```

Robot有一个name属性和一个ArrayList——movements。其go()方法负责遍历每个移动并打印详细信息。其他两个类——ForwardMove 和LeftTurn——各有一个属性。虽然ForwardMove类只有一个名为dist的属性，但是在本节开头的代码中，已经将speed和duration两个属性指派给了left节点。工厂负责处理这些属性，一会将会介绍。



看一下这些工厂。FactoryBuilderSupport依赖于Factory接口。该接口提供了一些方法，分别用于控制节点的创建、处理节点属性的设置、设置节点间的父子关系以及确定该节点是否为叶节点等。Groovy中提供了Factory的一个默认实现，叫做AbstractFactory，如下所示：

```
// AbstractFactory.java代码片段，来自Groovy中的部分实现
public abstract class AbstractFactory implements Factory
{
    public boolean isLeaf() { return false; }

    public boolean onHandleNodeAttributes(FactoryBuilderSupport builder,
                Object node, Map attributes ) { return true; }

    public void onNodeCompleted(FactoryBuilderSupport builder,
```

```
                Object parent, Object node ) { }

    public void setParent(FactoryBuilderSupport builder,
                Object parent, Object child ) { }

    public void setChild(FactoryBuilderSupport builder,
                Object parent, Object child ) { }
}
```

isLeaf()的默认实现会返回false，说明该节点可以有一个处理子节点的闭包。

onHandle NodeAttributes()之中适合对属性执行任何特殊的处理，像left中的duration和speed。在这个方法内，我们将从attributes中去除任何已经处理过的属性。如果像默认实现中那样返回true，FactoryBuilderSupport会将attributes中找到的剩余属性填充到节点实例中去。onNode Completed()方法会在节点处理完成时调用，而且在节点创建结束时可以执行一些最终操作。setParent()会在子节点的工厂上调用，所以可以设置任何父子关系。类似地，setChild()会在父节点的工厂上调用。AbstractFactory中唯一没有提供的Factory中的方法是newInstance()，它负责实例化实际的节点。

这个例子中，Robot、ForwardMove和LeftTurn分别需要一个工厂，它们对应的Robot-Factory、ForwardMoveFactory和LeftTurnFactory如下：

**UsingBuilders/UsingFactoryBuilderSupport.groovy**

```
class RobotFactory extends AbstractFactory {
  def newInstance(FactoryBuilderSupport builder, name, value, Map attributes ) {
      new Robot(name: value)
  }

  void setChild(FactoryBuilderSupport builder, Object parent, Object child) {
    parent.movements << child
  }
}

class ForwardMoveFactory extends AbstractFactory {
  boolean isLeaf() { true }
  def newInstance(FactoryBuilderSupport builder, name, value, Map attributes ) {
    new ForwardMove()
  }

  boolean onHandleNodeAttributes(FactoryBuilderSupport builder,
                Object node, Map attributes) {
    if (attributes.speed && attributes.duration) {
        node.dist = attributes.speed * attributes.duration
        attributes.remove('speed')
        attributes.remove('duration')
    }
```

```
    true
  }
}

class LeftTurnFactory extends AbstractFactory {
  boolean isLeaf() { true }

  def newInstance(FactoryBuilderSupport builder, name, value, Map attributes) {
    new LeftTurn()
  }
}
```

每个工厂的newInstance()方法负责实例化相应节点。在RobotFactory的setChild()方法中，向Robot的movements列表中添加了移动节点。因为forward和left是叶节点，其工厂中的isLeaf()方法会返回true。ForwardMoveFactory的onHandleNodeAttributes()中提供了对forward节点的特殊属性的支持。

花一分钟看一下isLeaf()方法带来的好处。在下面的例子中，我们向forward节点提供了一个闭包：

**UsingBuilders/UsingFactoryBuilderSupport.groovy**

```
def robotBldr = new RobotBuilder()
robotBldr.robot('bRobot') {
  forward(dist: 20) { }
}
```

FactoryBuilderSupport类意识到forward节点不能有嵌套的层次，于是作出拒绝，如下所示：

```
java.lang.RuntimeException: 'forward' doesn't support nesting.
```

要实现处理多个明确定义的节点的生成器，使用FactoryBuilderSupport要比使用BuilderSupport清晰很多。FactoryBuilderSupport还提供了其他便捷方法，可以拦截节点创建的生命周期，因此，如果需要的话，可以对节点遍历施加更多控制，比如可以使用preInstantiate()方法在工厂创建节点之前执行一些动作，或者是覆盖postNodeCompletion()方法，在节点处理完毕之后执行一些动作。如果需要在构建时执行其他任务，可以使用诸如FactoryBuilderSupport的getCurrentNode()和getParentNode()等便捷方法，轻松地处理正在创建的层次式结构。关于生成器及其API的更多细节，参见http://groovy.codehaus.org/FactoryBuilderSupport和http://groovy.codehaus.org/api/groovy/util/Factory BuilderSupport.html。

本章介绍了如何使用Groovy的生成器。对于一些枯燥的任务，比如创建XML或HTML文档，生成器提供了用于执行它们的DSL语法。可以使用Groovy提供的生成器，也可以亲自创建生成器。而且如果我们创建了一个有用的生成器，应该考虑将其贡献给社区！

我们意识并体会到了Groovy之强大。Groovy的动态特性需要我们的自律——对代码进行自动化测试非常重要。下一章将探索如何编写单元测试。

第 18 章

# 单元测试与模拟

18

单元测试对于元编程至关重要。不管静态类型语言的编译器执行的类型检查多么弱，动态类型语言甚至连这种程度的支持都没有。这就是动态语言中单元测试存在的必要原因了。（参见*Test Driven Development: By Example* [Bec02]、*Pragmatic Unit Testing in Java with JUnit* [TH03]和*JUnit Recipes: Practical Methods for Programmer Testing* [Rai04]。）尽管在动态语言中可以轻松地利用动态能力和元编程，但我们必须花些时间，确保程序的表现符合我们的预期，而不仅仅是符合我们的输入——要知道输入也会犯错。

开发者对单元测试的认识已经较几年前有所进步，遗憾的是，其应用却还不够。单元测试类似于对软件进行锻炼，它能够改进代码的健康度。这一点大部分开发者都同意，然而他们总有种种不做单元测试的借口。

单元测试不只是Groovy编程的关键，在Groovy中进行单元测试也很容易，而且充满乐趣。JUnit内置到了Groovy中。元编程能力使得创建模拟对象非常容易。Groovy还有一个内置的模拟库。接下来看一下如何使用Groovy对Java和Groovy应用进行单元测试。

## 18.1 本书代码与自动化单元测试

我所介绍的单元测试并不是抽象的建议。本书中的全部代码都使用了自动化单元测试。这是因为我在使用的是一门一直在演进的语言，Groovy的特性会改变，实现也会改变，bug会被修复，新特性会加入进来，变化不一而足。就在写作这些章节、编写代码示例时，我机器上安装的Groovy更新了好多次。如果某次更新后，因为某个特性或实现的变更，一个代码示例无法工作了，我就得及时发现并理解其原因，不能花掉太多精力。此外，随着写作的进行，我重构了书中的一些例子。有了自动化单元测试，我知道在经历了语言更新和我自己的重构之后，书中的例子仍然能够工作，这样我就睡得更踏实了。

在编写了最初的一些示例不久之后，我决定休息一下，想办法将所有示例的测试自动化，同时保持其独立性，并且可以放在隔离的文件中。有些示例是函数，还有一些是独立的程序或脚本。有了Groovy的元编程能力，结合`ExpandoMetaClass`以及加载并执行脚本的能力，创建和执行自动化单元测试轻而易举。

我花了几个小时才知道该怎么做。每当我写完一个新的示例，我就花最多二分钟的时间为这个示例写好测试。没几天，投入的精力和时间就初见成效了，后来效果又好了几倍。随着我更新Groovy，有些示例会运行失败。更重要的是，这些测试能够确保其他的例子正常工作，而且是有效的。

这些测试至少给我带来了5个方面的好处：

- 促进了我对Groovy特性的理解；
- 在Groovy用户邮件列表中提出的问题帮助修复了一些Groovy的bug；
- 帮助找到并修复了Groovy文档中的一处不一致；
- 帮我确保所有的示例都是有效的，而且在最新版的Groovy中可以很好地工作；
- 让我有勇气随意、随时重构任何示例，而且让我可以充满自信地说：我的重构会改进代码的结构，但是不会影响预期行为 。

## 18.2 对 Java 和 Groovy 代码执行单元测试

因为出色的Java-Groovy集成，任何基于Java的测试框架和模拟对象框架，如EasyMock、JMock和Mockito等，都可以结合Groovy使用。而且还不止这些，在安装Groovy时，已经自动获得了一个构建于JUnit之上的单元测试框架。可以使用它来测试Java虚拟机上的任何代码，包括Java代码、Groovy代码，等等。只需要从`GroovyTestCase`扩展出自己的测试类，并实现测试方法，然后准备运行测试就可以了。

**单元测试必须满足FAIR条件**

在编写单元测试时，要记住测试必须满足FAIR条件，这些条件是快速（fast）、自动化（automated）、隔离（isolated）和可重复（repeatable）。

测试必须要快。随着代码的演进和重构，需要快速得到代码仍然满足预期的反馈。如果测试很慢，开发者势必不愿费劲运行它们。因此需要一个非常快速的编辑和运行周期。

测试必须是自动化的。手工测试很累人，而且容易出错，也相当于减少了投入在重要任务上的时间。自动化测试就像立在我们肩上的天使，写代码的时候它们会安静地注视；如果代码不符合预期，它们会在我们的耳畔低语。它们会在代码开始走向崩溃的早期提供反馈。有一点可能都认同，我们宁可听计算机说我们的代码很糟糕，也不愿意听到同事这么说。自动化测试会让我们看上去很优秀，而且可以信赖。当我们说干完了的时候，我们知道代码会按预期方式工作。

测试必须隔离。如果碰到1031个编译错误，通常问题就是少了个分号，是吧？这没有实质性的帮助，一个小错误级联到一堆报告的错误是毫无意义的。我们想要的是隐藏的bug或错误与失败的测试用例之间的直接关联。这才能帮助我们快速找到并修复问题，而不是用大

量的失败测试把我们吓倒。隔离确保了一个测试不会遗留下可能会影响另一个测试的残余状态，从而就可以以任何顺序运行测试，可以运行所有测试、单个测试或是选定的一些测试。

测试必须是可以重复的。测试一定要能够运行任意多次，并且得到的是确定性的、可预测的结果。如果这次运行失败，未做任何修改，下次运行就通过了，这种测试是最坏的。线程原因可能会导致一些这种问题。再举一个例子，如果一个测试是向数据库中插入数据，而且存在唯一的列约束，那随后再运行同样的测试而不清理数据库，则测试会失败。不过如果该测试会回滚事务，这种情况就不会出现，该测试就是可重复的。测试的可重复性对于在快速演进代码的同时保持清醒非常关键。

先编写一个简单的测试：

**UnitTestingWithGroovy/ListTest.groovy**

```
class ListTest extends GroovyTestCase {
  void testListSize() {
    def lst = [1, 2]
    assertEquals "ArrayList size must be 2", 2, lst.size()
  }
}
```

即使Groovy是动态类型的，JUnit还是期望测试方法的返回类型为`void`。这意味着在定义测试方法时，必须显式地使用`void`代替`def`。Groovy的可选类型在这起到了帮助。要运行前面的代码，只需要像执行任何Groovy程序那样执行它。输入下列命令：

```
groovy ListTest
```

输出如下：

```
.
Time: 0.006

OK (1 test)
```

如果熟悉JUnit，这里的输出也就理解了——一项测试成功执行。

如果喜欢看到红绿条，可以在支持运行测试的集成开发环境（IDE）内运行单元测试。

也可以调用`junit.swingui.TestRunner`的`run()`方法，并将Groovy测试类的名字传给它，这样运行测试就能在Swing GUI内看到红绿条了。

可以使用JUnit中我们所熟悉的任何assert方法。为方便使用，Groovy添加了更多`assert`方法，且举几种，有`assertArrayEquals()`、`assertLength()`、`assertContains()`、`assertToString()`、`assertInspect()`、`assertScript()`和`shouldFail()`等。

在编写单元测试时，考虑编写3种类型的测试：正面测试、负面测试和异常测试。正面测试

可以帮助确定代码的表现符合预期。可以在正常的路径上调用这种测试。比如，在一个账户中存入100美元，然后检查余额是不是多了这么多。负面测试检查的是，代码能否按预期方式处理前置条件失效、无效输入等问题。存入一个负值，看看代码会做什么。如果账户关闭，又会怎么样呢？异常测试可以帮助确定的是，当异常情况出现时，代码是否会抛出正确的异常，以及表现是否符合预期。如果账户关闭后，自动取款操作发起，会怎么样？在这一点上一定要相信我，我遇到过一个"有创意"的银行就出现过这种场景。像这样的情况可以受益于异常测试。

以这些术语来思考测试，有助于把实现的逻辑彻底想清楚。我们不仅要处理实现逻辑的代码，还要考虑常常会让我们陷入困境的边界条件和极端条件。

利用Groovy和JUnit提供的`assert`方法，可以轻松实现正面测试。实现负面测试和异常测试需要的工作会多一点，Groovy也有可以提供帮助的机制，下一节将会介绍。

即使主项目是用Java写的，也应该考虑用Groovy编写测试代码。因为Groovy是轻量级的，我们会发现，在Groovy中编写测试会更容易、更快、更有乐趣。这也是在以Java为主的项目中实践Groovy的一种不错的方式。

假设在src目录下有一个Java类`Car`，如以下代码所示。再假设我们已经使用`javac`将其编译到了classes目录下。

**UnitTestingWithGroovy/src/Car.java**

```
// Java代码
package com.agiledeveloper;

public class Car
{
  private int miles;
  public int getMiles() { return miles; }
  public void drive(int dist)
  {
    miles += dist;
  }
}
```

可以在Groovy中为这个类编写单元测试，而且不必编译测试代码来运行它。下面是`Car`类的一些正面测试。这些测试都位于test目录下的CarTest.groovy文件中。



**UnitTestingWithGroovy/test/CarTest.groovy**

```
class CarTest extends GroovyTestCase
{
  def car
  void setUp()
  {
    car = new com.agiledeveloper.Car()
  }
```

```
  void testInitialize()
  {
    assertEquals 0, car.miles
  }
  void testDrive()
  {
    car.drive(10)
    assertEquals 10, car.miles
  }
}
```

setUp()方法和相应的tearDown()方法（前面的例子中没有显示出来）会将每个测试调用夹在中间。可以在setUp()中初始化对象，可以根据需要在tearDown()中执行清理或复位操作。这2个方法可以帮助避免复制代码，同时可以帮助将测试彼此隔离起来。

要运行该测试，请输入groovy -classpath classes test/CarTest命令。应该会看到如下输出：

```
..
Time: 0.003

OK (2 tests)
```

输出表明两项测试都执行了，而且不出意外，均成功通过。第一项测试确认了该Car一开始里程表上的公里数为0；驾驶一定的距离，里程表上会增加相应的公里数。现在编写一个负面测试：

```
void testDriveNegativeInput()
{
  car.drive(-10)
  assertEquals 0, car.miles
}
```

drive()的参数使用了负值——-10。我们判断，这种情况下Car一定会忽略我们的驾驶请求，所以预计里程数不会改变。然而Java代码没有处理这一条件，没有检查输入参数就修改了里程数。运行之前的测试，会出现一项失效：

```
...F
Time: 0.004
There was 1 failure:
1) testDriveNegativeInput(CarTest)
  junit.framework.AssertionFailedError:
  expected:<0> but was:<-10>

...

FAILURES!!!
Tests run: 3,  Failures: 1,  Errors: 0
```

输出表明，两个正面测试通过，但是负面测试失败了。可以修复Java代码来正确地处理这种情况。可见，使用Groovy测试Java代码相当直接，而且简单。

## 18.3　测试异常

现在看一下编写异常测试。可以将要测试的方法包在try-catch中。如果该方法抛出了预期的异常，也就是说进入了catch块，则一切正常。

如果代码没有抛出任何异常，会调用fail()来说明测试失败，如下所示：

**UnitTestingWithGroovy/ExpectException.groovy**

```
try {
  divide(2, 0)
  fail "Expected ArithmeticException ..."
} catch(ArithmeticException ex) {
  assertTrue true // 成功
}
```

前面的代码是Java风格的JUnit测试，在Groovy中也可以使用。不过，Groovy提供了一个shouldFail()方法，它可以优雅地将样板代码包起来，这使得编写异常测试更容易了。使用它来编写一个异常测试：

**UnitTestingWithGroovy/ExpectException.groovy**

```
shouldFail { divide(2, 0) }
```

shouldFail()方法接受一个闭包。它会在一个看守式的try-catch块中调用该闭包。如果没有抛出异常，它会通过调用fail()方法抛出一个异常。如果意在于捕获一个具体的异常，可以向shouldFail()方法指明这一信息：

**UnitTestingWithGroovy/ExpectException.groovy**

```
shouldFail(ArithmeticException) { divide(2, 0) }
```

在这个例子中，shouldFail()期望闭包抛出ArithmeticException。如果代码抛出了ArithmeticException，或者抛出的是扩展了该异常的某些异常类，测试正常通过。如果抛出的是其他某些异常，或者没有抛出异常，则shouldFail()失败。我们可以利用Groovy中括号的灵活性（参见19.9节），将前面的调用写成下面这样：

**UnitTestingWithGroovy/ExpectException.groovy**

```
shouldFail ArithmeticException, { divide(2, 0) }
```

## 18.4　模拟

要对存在依赖的大片代码进行单元测试，就算并非没有可能，也是非常困难的。（怎么算大呢？在编辑器内，如果不往下滚动就没法看全，这就算大了——嘿，别忙着去调小字体哦。）单

元测试有一个优点，它会强制我们让代码单元小一些。代码越小，内聚性就越高。它还会强制我们将代码与其周边环境解耦，也就意味着降低耦合性。高内聚和低耦合是优秀设计的特质。本节将探讨处理依赖的不同方式，后面几节则将探讨存在依赖时的单元测试。

耦合有两种形式：有的代码会依赖我们的代码，有的代码是我们的代码所依赖的。在对我们的代码进行单元测试之前，需要解决这两种耦合。

必须将被测代码从其所在的应用中分离或解耦出来。假设有逻辑位于GUI内按钮的事件处理器中，这种逻辑很难进行单元测试。因此，为进行单元测试，必须将其代码分离到一个方法中。

再假设有逻辑严重依赖某个资源。那个资源可能响应很慢，使用代价高昂，不可预测，或者目前正处于开发之中。因此，在有效地进行单元测试之前，必须将这种依赖从代码中分离出去。这可以借助存根和模拟来实现。

**存根与模拟的对比**

存根用于代替真正的对象。当被测方法调用存根时，它会根据设定好的预期响应简单地应答。之所以要设置响应，是为了满足测试通过的需要。模拟对象所做的事情要比存根多得多。它可以帮助确定被测代码按预期方式与其依赖或协作对象交互。模拟对象可以记录代码中在该对象所代表的协作对象上进行的方法调用的次序和次数。它可以确保传递给方法调用的是正确的参数。存根验证状态，而模拟验证行为。当在测试中使用模拟时，它不仅验证状态，也验证代码与其依赖的交互行为[1]。

Groovy同时支持创建存根和模拟，18.10节将会介绍。

我们的代码所依赖的代码被称作协作者，我们的代码与协作者合作完成其工作。协作者可以是一个组件、一个对象、一个层次或一个子系统。它可能是局部的，可能位于我们的对象内部，可能是作为参数出入的，甚至可能是远程的。没有协作者，我们的对象就无法行使其功能。然而为了满足测试需求，需要替换掉协作者。

模拟用于代替协作者，它不做任何真正的工作，而只是简单地向调用它的代码做出预期响应，以便让测试工作。

当运行应用时，我们希望代码依赖的是它所需要的真正对象（协作者）。进行集成测试时也是如此。然而在进行单元测试时，我们希望代码依赖模拟。因此，需要想个办法，通过开关控制我们的代码依赖模拟，还是依赖真正的对象。

① tin Fowler在“Mocks Aren’t Stubs”（模拟并非存根）一文中探讨了存根与模拟的不同，参见http://martinfowler.com/articles/mocksArentStubs.html。

在像Java这样的静态类型语言中，可以使用接口实现，如图18-1所示。Java中的框架（如EasyMock、JMock和Mockito等）简化了模拟，其中有一些甚至支持在不创建接口的条件下实现模拟。使用一种基于代理的机制，它们会拦截调用，并将请求路由给模拟对象，而不是真正的依赖对象。

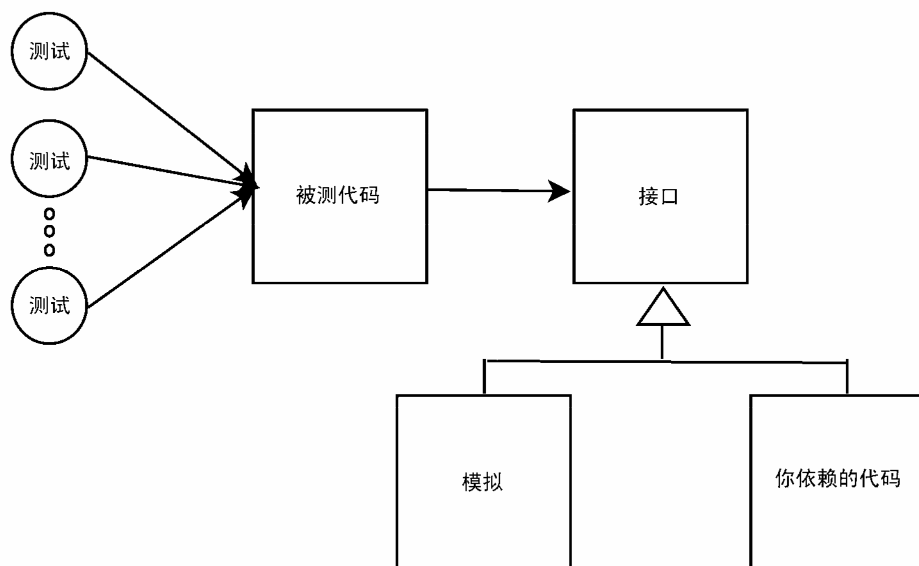



图18-1 单元测试中的模拟

Groovy的动态特性与元编程能力在这方面提供了很大的优势。在Groovy中有很多种创建模拟的方式。我们可以使用下列特性：

- 方法覆盖
- 分类
- `ExpandoMetaClass`
- `Expando`
- `Map`
- Groovy的Mock Library

下面几节将探讨在Groovy中创建和使用模拟的技术。



## 18.5 使用覆盖实现模拟

假设有一个类，它依赖一个做某个重要工作的方法，而这个方法在执行时要消耗大量的时间和资源，比如下面的`myMethod()`：

**UnitTestingWithGroovy/com/agiledeveloper/CodeWithHeavierDependencies.groovy**

```
package com.agiledeveloper

public class CodeWithHeavierDependencies
{
  public void myMethod()
  {
    def value = someAction() + 10

    println(value)
  }

  int someAction()
  {
    Thread.sleep(5000) // 模拟消耗时间的动作

    return Math.random() * 100 // 模拟某个动作的结果
  }
}
```

现在打算测试myMethod()（它从属于CodeWithHeavierDependencies）。然而该方法依赖于someAction()，后者模拟了消耗时间和资源的操作。

如果简单地为myMethod()编写一个单元测试，它会很慢。还有另外一个问题，无法对调用myMethod()的结果执行断言，因为它什么都没返回，只是将一个值打印到了标准输出。这里需要捕获它打印的值，并施以断言。所以这个方法难以测试：慢，而且复杂。

为解决这些问题，下面覆盖这个讨厌的方法：

**UnitTestingWithGroovy/TestByOverriding.groovy**

```
import com.agiledeveloper.CodeWithHeavierDependencies

class TestCodeWithHeavierDependenciesUsingOverriding extends GroovyTestCase {
  void testMyMethod() {
    def testObj = new CodeWithHeavierDependenciesExt()

    testObj.myMethod()

    assertEquals 35, testObj.result
  }
}

class CodeWithHeavierDependenciesExt extends CodeWithHeavierDependencies {
  def result

  int someAction() { 25 }

  def println(text) { result = text }
}
```

运行这段代码，并确保测试快速通过：

```
.
Time: 0.015

OK (1 test)
```

这段代码中创建了一个用于模拟的新类——CodeWithHeavierDependenciesExt，它扩展了CodeWithHeavierDependencies。新类模拟了someAction和println()方法。（Groovy的习惯是简单地用println()代替System.out.println()，这里利用了这种习惯，提供了一个局部的println()实现——明白了吧？）运行测试代码，看看能否成功。这次没有延迟，标准输出也没有乱糟糟的内容。

现在仍然在测试行为，但是通过将非确定性的行为变为确定性的，就能够针对它编写断言了。必须想出一个聪明的办法，将依赖模拟出来，这样就可以将精力集中到对我们代码的行为做单元测试上了。

前面的例子测试了Groovy类中的一个方法，也可以使用这种方式测试Java类。

通过覆盖Java类中的方法（如someAction()）来模拟，也不是问题。然而和Groovy代码调用println()不同，Java代码要调用System.out.println()。因此，在用于模拟的派生类中创建一个println()，起不到什么帮助。不过可以扩展PrintStream并替换System.out。下面看一个与之前测试的Groovy类等价的Java类：

**UnitTestingWithGroovy/com/agiledeveloper/JavaCodeWithHeavierDependencies.java**

```java
package com.agiledeveloper;

public class JavaCodeWithHeavierDependencies
{
  public int someAction()
  {
    try
    {
      Thread.sleep(5000); //模拟消耗时间的动作
    }
    catch(InterruptedException ex) {}

    return (int) (Math.random() * 100); //模拟某个动作的结果
  }

  public void myMethod()
  {
    int value = someAction() + 10;

    System.out.println(value);
  }
}
```

用于测试上述Java代码的Groovy代码如下：

**UnitTestingWithGroovy/TestJavaByOverride.groovy**

```
import com.agiledeveloper.JavaCodeWithHeavierDependencies

class TestCodeWithHeavierDependenciesUsingOverriding extends GroovyTestCase {
  void testMyMethod() {
    def testObj = new ExtendedJavaCode()

    def originalPrintStream = System.out
    def printMock = new PrintMock()
    System.out = printMock

    try {
      testObj.myMethod()
    } finally { System.out = originalPrintStream }

    assertEquals 35, printMock.result
  }
}

class ExtendedJavaCode extends JavaCodeWithHeavierDependencies {
  int someAction() { 25 }
}


class PrintMock extends PrintStream {
  PrintMock() { super(System.out) }

  def result

  void println(int text) { result = text }
}
```

输出符合预期，测试通过：

```
.
Time: 0.026

OK (1 test)
```

被测试的myMethod()方法是JavaCodeWithHeavierDependencies类的一部分。这里创建了ExtendedJavaCode类，它扩展了JavaCodeWithHeavierDependencies，并覆盖了someAction()方法。此外还创建了一个扩展了PrintStream的PrintMock类，并将System.out指派给了这个类的实例。这可以帮助拦截对System.out.println()调用，并直接转向我们的模拟实现。

## 18.6 使用分类实现模拟

13.1节探讨了分类可以在Groovy中提供可控的AOP。本节将介绍如何使用分类实现模拟。

**UnitTestingWithGroovy/TestUsingCategories.groovy**

```
import com.agiledeveloper.CodeWithHeavierDependencies

class TestUsingCategories extends GroovyTestCase {
  void testMyMethod() {
    def testObj = new CodeWithHeavierDependencies()

    use(MockHelper) {
      testObj.myMethod()

      assertEquals 35, MockHelper.result
    }
  }
}

class MockHelper {
  def static result

  def static println(self, text) { result = text }

  def static someAction(CodeWithHeavierDependencies self) { 25 }
}
```

MockHelper有两个静态方法，分别对应我们想模拟的方法——someAction()和println()。在这个测试内，通过使用use(MockHelper)，让分类来拦截对方法的调用，并在适当的情况下替换这两个方法。这很像AOP中使用的建议（Advice）。

这段代码让人放心地通过了测试，如下所示：

```
.
Time: 0.027

OK (1 test)
```

分类仅在Groovy代码中才有用，无助于模拟在编译好的Java代码中调用的方法。

上一节介绍的覆盖方式在Java和Groovy代码中均可使用。然而，如果被测试的类是final的，这种方式就无法使用了。而分类方式可以应用于那种情况。



## 18.7 使用 ExpandoMetaClass 实现模拟

在Groovy中还可以以另外一种方式拦截方法调用，也就是使用ExpandoMetaClass（参见13.2节和13.3节）。第13章中的两节介绍的两种方式都不需要创建一个单独的类。但是，可以为每个想要模拟的方法创建一个闭包，然后将其设置到被测试实例的MetaClass中。来看一个例子。

为被测试的实例创建一个ExpandoMetaClass实例。这个MetaClass将提供协作方法的模拟实现。

下面这段代码中，为模拟println()创建了一个闭包，并将其设置到了用于ClassWithHeavierDependencies的一个ExpandoMetaClass实例中，见第7行。类似地，这里也为模拟someAction()创建了一个闭包，见第8行。专门为被测试的实例创建一个ExpandoMetaClass实例，其优点是，不会对CodeWithHeavierDependencies的元类造成全局的影响。也就是说，如果还有其他测试，这里模拟的方法不会影响它们（别忘了，要保持测试彼此隔离）。

**UnitTestingWithGroovy/TestUsingExpandoMetaClass.groovy**

```
Line 1 import com.agiledeveloper.CodeWithHeavierDependencies
     -
     - class TestUsingExpandoMetaClass extends GroovyTestCase {
     -   void testMyMethod() {
     5     def result
     -     def emc = new ExpandoMetaClass(CodeWithHeavierDependencies, true)
     -     emc.println = { text -> result = text }
     -     emc.someAction = { -> 25 }
     -     emc.initialize()
    10
     -     def testObj = new CodeWithHeavierDependencies()
     -     testObj.metaClass = emc
     -
     -     testObj.myMethod()
    15
     -     assertEquals 35, result
     -   }
     - }
```

输出证实测试通过：

```
.
Time: 0.031

OK (1 test)
```

在这个例子中，当myMethod()调用println()和someAction()这2个方法时，ExpandoMetaClass会拦截这些调用，并将其路由到模拟实现。再次提醒，这与AOP中的建议很类似。

创建模拟，设定预期，然后将其应用于测试中，这些步骤在前面的例子中都很好地包含了。没有创建额外的类。如果有其他测试，可以简洁地创建必要的模拟，以满足那些测试的需要。

通过ExpandoMetaClass实现模拟的这种方式，只能在Groovy代码中使用；对于在编译好的Java代码内调用的方法，则没什么帮助。

## 18.8 使用 Expando 实现模拟

到目前为止，本章介绍的都是如何模拟在另一个实例方法内调用到的实例方法，接下来将介绍如何模拟我们的代码所依赖的其他对象。

先来看一个例子。假设想要测试某个类中的方法，这些方法依赖一个文件（File）。这使得单元测试很难编写。为了实现快速、自动化的单元测试，需要想办法模拟这个文件对象：

**UnitTestingWithGroovy/com/agiledeveloper/ClassWithDependency.groovy**

```
package com.agiledeveloper

public class ClassWithDependency
{
  def methodA(val, file)
  {
    file.write "The value is ${val}."
  }
  def methodB(val)
  {
    def file = new java.io.FileWriter("output.txt")
    file.write "The value is ${val}."
  }
  def methodC(val)
  {
    def file = new java.io.FileWriter("output.txt")
    file.write "The value is ${val}."
    file.close()
  }
}
```

代码中有三个方法，存在不同形式的依赖。methodA()接受一个实例，看上去像File。其他2个方法——methodB()和methodC()，则在内部实例化了一个FileWriter实例。Expando类只能帮助处理第一个方法。考虑到这一点，本节将只探讨methodA()的测试。至于如何测试另2个方法，则将在18.10节介绍。

methodA()使用File的write()方法向给定文件对象中写入了一条消息。我们想测试methodA()，但是不希望真把消息写入物理文件，之后再读回来用于断言。

这里可以利用Groovy的动态类型，因为methodA()没有指明参数的类型。因此，可以发送任何对象，只要它具备预期参数的能力，比如说提供了write()方法（参见3.4节）。现在就动手吧。创建一个包含write()方法的HandTossedFileMock类。不必担心真正的File类中的所有属性和方法，只需要关心被测试的方法实际调用到的。代码如下：

18

**UnitTestingWithGroovy/TestUsingAHandTossedMock.groovy**

```
import com.agiledeveloper.ClassWithDependency
class TestWithExpando extends GroovyTestCase {
  void testMethodA() {
    def testObj = new ClassWithDependency()
    def fileMock = new HandTossedFileMock()
    testObj.methodA(1, fileMock)

    assertEquals "The value is 1.", fileMock.result
  }
}

class HandTossedFileMock {
  def result
  def write(value) { result = value }
}
```

这段代码的输出证实测试通过：

```
.
Time: 0.015

OK (1 test)
```

在这段代码中，HandTossedFileMock类的模拟实现write()，只是简单地将接受的参数保存到一个result属性中。用这个模拟类的实例代替真正的File，发送给methodA()。多亏了动态类型特性，methodA()能够非常开心地使用这个模拟实例。

实现并不难，然而如果不必弄出一个单独的类，那就更好了。这就该Expando大显身手了（参见15.1节）。

只需创建一个Expando实例，为其提供一个text属性和一个write()方法的模拟实现。然后将这个实例传递给methodA()。看一下代码：

**UnitTestingWithGroovy/TestUsingExpando.groovy**

```
import com.agiledeveloper.ClassWithDependency

class TestUsingExpando extends GroovyTestCase {
  void testMethodA() {
    def fileMock = new Expando(text: '', write: { text = it })

    def testObj = new ClassWithDependency()
    testObj.methodA(1, fileMock)
    assertEquals "The value is 1.", fileMock.text
  }
}
```

输出如下：

```
.
Time: 0.022

OK (1 test)
```

在前面的两个例子中，调用methodA()方法时都没有创建真正的物理文件。单元测试运行很快，而且测试完毕后我们不用读取或清理任何文件。

当向被测试方法传递依赖的对象时，Expando很有用。然而，如果方法会在内部创建依赖的对象（比如methodB()和methodC()），它就于事无补了。18.10节将解决这类问题。

## 18.9　使用 Map 实现模拟

上一节介绍了一个使用Expando实现模拟对象的例子，其实也可以使用Map。众所周知，Map中有键和与键关联的值。其中的值可以是对象，甚至可以是闭包。利用这一点，可以使用一个Map来代替协作对象。

将上一节使用Expando的例子，使用Map重写：

**UnitTestingWithGroovy/TestUsingMap.groovy**

```
import com.agiledeveloper.ClassWithDependency

class TestUsingMap extends GroovyTestCase {
  void testMethodA() {
    def text = ''
    def fileMock = [write : { text = it }]

    def testObj = new ClassWithDependency()
    testObj.methodA(1, fileMock)

    assertEquals "The value is 1.", text
  }
}
```

输出如下：

```
.
Time: 0.029

OK (1 test)
```

与Expando类似，当向被测方法传递依赖的对象时，Map也很有用。但如果协作对象是在被测方法内部创建的，它就帮不上什么忙了。下一节会解决这个问题。

## 18.10　使用 Groovy Mock Library 实现模拟

Groovy Mock Library是在groovy.mock.interceptor包中实现的，用于模拟较深的依赖，

也就是模拟被测方法内创建的协作对象或依赖对象。StubFor和MockFor是负责这一功能的两个类。下面依次看一下。

StubFor和MockFor的意图是像分类那样拦截方法调用（参见18.6节）。然而与分类不同的是，不必为模拟创建单独的类。在使用时，只需要在StubFor或MockFor的实例上引入模拟方法，这些类会负责替换我们要模拟的对象的MetaClass。

18.4节探讨了存根和模拟的不同。下面就先从一个使用StubFor的例子入手，理解存根的优势与不足；之后再使用MockFor，看一下模拟的优势。

## 18.10.1　使用StubFor

使用Groovy的StubFor为File类创建存根：

**UnitTestingWithGroovy/TestUsingStubFor.groovy**

```
Line 1 import com.agiledeveloper.ClassWithDependency
     -
     - class TestUsingStubFor extends GroovyTestCase {
     -   void testMethodB() {
     5     def testObj = new ClassWithDependency()
     -
     -     def fileMock = new groovy.mock.interceptor.StubFor(java.io.FileWriter)
     -     def text
     -     fileMock.demand.write { text = it.toString() }
    10     fileMock.demand.close {}
     -
     -     fileMock.use {
     -       testObj.methodB(1)
     -     }
    15
     -     assertEquals "The value is 1.", text
     -   }
     - }
```

在创建StubFor的实例时，首先提供想为其创建存根的类——这里是java.io.FileWriter。之后为write()方法的存根实现创建一个闭包。第12行在该存根上调用了use()方法。此时，它会将FileWriter的MetaClass替换为一个ProxyMetaClass。在所附的闭包内，对FileWriter实例的任何调用都会被路由到该存根。

然而存根和模拟不会帮助拦截对构造器的调用。在上述例子中，FileWriter的构造器被调用了，结果是磁盘上创建了一个名为output.txt的文件。

StubFor帮助我们测试的是，methodB()方法是否创建了一个正常的FileWriter实例，并将期望的内容写到了这个实例中。不过它是有局限性的。它没有测试当关闭文件时，这个方法的表

现是否正常。即使在存根上要求了close()方法（参见代码第10行），它也不会检查该方法是不是真被调用了。存根只是简单地代替协作者并验证状态。要验证行为，必须使用模拟，具体而言，就是使用MockFor类。

### 18.10.2 使用MockFor

对前面的测试代码做一处修改：

**UnitTestingWithGroovy/TestUsingMockFor.groovy**

```
//def fileMock = new groovy.mock.interceptor.StubFor(java.io.FileWriter)
def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
```

将StubFor替换为MockFor，这是唯一的修改。现在运行测试，输出如下：

```
.F
Time: 0.093
There was 1 failure:
1) testMethod1(TestUsingStubFor)junit.framework.AssertionFailedError:
verify[1]: expected 1..1 call(s) to 'close' but was never called.
```

与存根不同，模拟会指出，纵然代码产生了指定结果， 但是其表现与预期不符。也就是说，这段代码没有调用使用demand在测试预期中设置的close()方法。

methodC()所做的事情与methodB()相同，但是它调用了close()。使用MockFor测试这个方法：

**UnitTestingWithGroovy/TestMethodCUsingMock.groovy**

```
import com.agiledeveloper.ClassWithDependency

class TestMethodCUsingMock extends GroovyTestCase {
  void testMethodC() {
    def testObj = new ClassWithDependency()

    def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
    def text
    fileMock.demand.write { text = it.toString() }
    fileMock.demand.close {}
    fileMock.use {
      testObj.methodC(1)
    }
    assertEquals "The value is 1.", text
  }
}
```

在这种情况下，模拟会告诉我们，与协作者的合作非常愉快。测试通过，输出如下：

```
.
Time: 0.088

OK (1 test)
```

前面几个例子中，被测方法仅创建了一个被模拟对象FileWriter的实例。如果该方法会创建多个实例，又会怎么样呢？这个模拟就会代表所有实例，但是必须为每个实例创建预期要求，也就是要通过demand有所体现。来看一个使用了两个FileWriter实例的例子。下列代码中的useFiles()方法将给定参数复制到第一个文件中，并将参数的大小写到第二个文件中：

```
class TwoFileUser {
  def useFiles(str) {
    def file1 = new java.io.FileWriter("output1.txt")
    def file2 = new java.io.FileWriter("output2.txt")
    file1.write str
    file2.write str.size()
    file1.close()
    file2.close()
  }
}
```

相应的测试代码如下：

**UnitTestingWithGroovy/TwoFileUserTest.groovy**

```
class TwoFileUserTest extends GroovyTestCase {
  void testUseFiles() {
    def testObj = new TwoFileUser()
    def testData = 'Multi Files'
    def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
    fileMock.demand.write() { assertEquals testData, it }
    fileMock.demand.write() { assertEquals testData.size(), it }
    fileMock.demand.close(2..2) {}
    fileMock.use {
      testObj.useFiles(testData)
    }
  }
  void tearDown() {
    new File('output1.txt').delete()
    new File('output2.txt').delete()
  }
}
```

运行这个测试，输出如下：

**UnitTestingWithGroovy/TwoFileUserTest.output**

```
.
Time: 0.091

OK (1 test)
```

这里创建的测试预期要求要由被测方法中创建的两个对象共同满足。模拟可以灵活地支持多个对象。当然，如果有大量的对象需要创建，模拟实现起来也很困难。可以利用下文将探讨的指

定多次调用功能。

对于模拟FileWriter类的方法，MockFor游刃有余；但是对于构造器，它却无法阻止其运行。因此，非常遗憾，在运行测试时，会有两个名字分别为output1.txt和output2.txt的空文件被创建出来。tearDown()方法对此做了清理。

模拟记录了一个方法被调用的次序和次数，如果被测代码没有严格满足所要求的测试预期，模拟将抛出一个异常，致使测试失败。

对于同一方法的多次调用，可以轻松地设定测试预期。这有一个例子：

```
def someWriter() {
  def file = new FileWriter('output.txt')
  file.write("one")
  file.write("two")
  file.write(3)
  file.flush()
  file.write(file.getEncoding())
  file.close()
}
```

假设只想测试我们的代码与协作者之间的交互，需要为如下操作设定测试预期：依次调用3次write()，1次flush()，1次getEncoding()，1次write()，最后再调用1次close()。

可以在demand中使用范围来指定一个调用的基数或重数。比如mock.demand.write(2..4){...}的意思是，预期该方法至少被调用2次，但最多4次。可以用这种方式为前面的方法编写一个测试，看看表达对多次调用和返回值的预期，以及对接受到的参数值是否符合预期施加断言是多么容易。

```
void testSomeWriter() {
  def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
  fileMock.demand.write(3..3) {} // 如果想表达最多3次，使用0..3
  fileMock.demand.flush {}
  fileMock.demand.getEncoding { return "whatever" } // return是可选的
  fileMock.demand.write { assertEquals 'whatever', it.toString() }
  fileMock.demand.close {}

  fileMock.use {
    testObj.someWriter()
  }
}
```

在这个例子中，模拟会断言write()被调用3次；然而它未能对传入的参数施加断言。可以修改其代码，以断言参数，如下所示：

```
def params = ['one', 'two', 3]
def index = 0
fileMock.demand.write(3..3) { assert it == params[index++] }
  // 如果想表达最多3次，使用0..3
```

本章介绍了Groovy库中内置的单元测试和模拟设施。某些强大和漂亮的第三方库使单元测试变得更容易、更有趣味了。对于模拟，可以看看gmock[①]。测试工具Spock带来了更高的流畅度，编写和表达单元测试也更容易了[②]。在Spock中，可以像`expect: expected == expression`这样简单地编写断言，可以提供一个包含输入值和预期值的数据表格，轻松地创建模拟。

单元测试需要很大的自律，然而收益大于成本。与静态类型语言相比，动态类型语言提供了更大的灵活性，所以单元测试非常关键。

本节介绍了使用存根和模拟管理依赖的技术。可以使用Groovy对Java代码进行单元测试。可以使用现有的单元测试和模拟框架，并且通过覆盖方法来模拟Groovy和Java代码。要对Groovy代码进行单元测试，可以使用分类和`ExpandoMetaClass`。二者都支持通过拦截方法调用实现模拟。而且如果使用`ExpandoMetaClass`，我们不必创建额外的类，测试会很简洁。

对于参数对象的简单模拟，可以使用`Map`或`Expando`。如果想对多个方法设置测试预期，以及想模拟被测方法内部的依赖，可以使用`StubFor`。要测试状态以及行为，则可以使用`MockFor`。

我们看到了Groovy的动态特性结合其元编程能力，是如何使单元测试变成一件乐事的。随着代码的演进、重构，以及我们对应用需求理解的加深，使用Groovy进行单元测试，可以帮助我们保持较快的开发速度。因为单元测试让我们有这种自信——我们的应用会继续满足预期。随着应用开发复杂性的提升，可以将单元测试当作一个安全钩。

① http://code.google.com/p/gmock/

② http://code.google.com/p/spock

# 第 19 章 在Groovy中创建DSL

领域特定语言（Domain-Specific Language，DSL）针对的是“某一特定类型的问题”，可以参考附录A中引用的Martin Fowler有关DSL的讨论。其语法聚焦于指定的领域或问题。我们不会像使用Java、Groovy或C++等语言那样将其应用于一般性编程任务，因为DSL的应用领域和能力都非常有限。

一门DSL会很小，很简单（尽管设计起来可能并不简单），很有表现力，聚焦于一个问题区域或领域。DSL有两大特点：上下文驱动，非常流畅。

DSL由来已久。很有可能我们已经接触过它了，比如在与外部应用通信时使用的某种特殊的关键字输入文件。Ant和Gant（参见附录A）也是DSL的例子。具体来说，Gant是一个Ant的包装器，它使用Groovy代替了XML，用于指定构建任务。

Groovy的动态特性与元编程能力，使其对于构建DSL很有吸引力。本章将探讨DSL，以及如何使用Groovy构建DSL。

## 19.1 上下文

上下文（Context）是DSL的特点之一。上下文也是人类赖以交流的基础，人之所以能够高效交流，上下文在谈话中所提供的连续性功不可没。前几天，听到我的朋友Neal 喊道：“Venti latte with two extra shots!”[1]。他这里用的就是星巴克的DSL。他完全没提到咖啡，但毫无疑问，他会得到一份，而且是价格较高的。这就是上下文驱动。

下面看看订购比萨的Java代码。这段代码缺乏上下文。引用`joesPizza`会被重复使用：

**CreatingDSLs/OrderPizza.java**

```
//Java代码
package com.agiledeveloper;
```

① 超大杯拿铁再加两份浓缩咖啡。星巴克的饮品按容量划分为中杯（Tall）、大杯（Grande）和超大杯（Venti）三种杯型。——译者注

```
public class OrderPizza {
  public static void main(String[] args) {
    PizzaShop joesPizza = new PizzaShop();
    joesPizza.setSize(Size.LARGE);
    joesPizza.setCrust(Crust.THIN);
    joesPizza.setTopping("Olives", "Onions", "Bell Pepper");
    joesPizza.setAddress("101 Main St., ...");
    int time = joesPizza.setCard(CardType.VISA, "1234-1234-1234-1234");
    System.out.printf("Pizza will arrive in %d minutes\n", time);
  }
}
```

因为有了`with()`方法（参见7.1节），使用Groovy编写的同功能代码就没这么杂乱：

**CreatingDSLs/OrderPizza.groovy**

```
import com.agiledeveloper.*

PizzaShop joesPizza = new PizzaShop()
joesPizza.with {
    setSize(Size.LARGE)
    setCrust(Crust.THIN)
    setTopping("Olives", "Onions", "Bell Pepper")
    setAddress("101 Main St., ...")
    int time = setCard(CardType.VISA, "1234-1234-1234-1234")
    printf("Pizza will arrive in %d minutes\n", time)
}
```

因为在Groovy中类型是可选的，括号也几乎总是可选的（参见19.9节），所以前面的代码可以更轻巧一些：

**CreatingDSLs/OrderPizza2.groovy**

```
import com.agiledeveloper.*

PizzaShop joesPizza = new PizzaShop()
joesPizza.with {
    setSize Size.LARGE
    setCrust Crust.THIN
    setTopping "Olives", "Onions", "Bell Pepper"
    setAddress "101 Main St., ..."
    time = setCard(CardType.VISA, "1234-1234-1234-1234")
    printf "Pizza will arrive in %d minutes\n", time
}
```

上下文使一切变得更紧凑了（好的方面），也减少了混乱，提升了实际效果。

## 19.2　流畅

流畅是DSL的另一特点。它改进了代码的可读性，同时使代码自然地流动。其实要在设计上

实现流畅并不容易，但我们应该让用户使用起来很流畅。下面将探讨一些与流畅有关的例子，并探索几种在Groovy中编写循环的方式：

**CreatingDSLs/FluentLoops.groovy**

```
// 传统循环
for(int i = 0; i < 10; i++) {
  println(i);
}
// Groovy方式
for(i in 0..9) { println i }

0.upto(9) { println it }

10.times { println it }
```

上述所有循环都会产生同样的结果。在众多特性之中，Groovy为循环提供了流畅性。然而，流畅性并非Groovy专有。在Java中，EasyMock（Groovy的模拟库就是受其启发）在设置模拟的预期表现时就表现出了流畅性：

```
//Java代码
      expect(alarm.raise()).andReturn(true);
      expect(alarm.raise()).andThrow(new InvalidStateException());
```

这段代码表明，在第一个调用上，`alarm`模拟对象应该返回true，而在第二个调用上则是抛出一个异常。

在Grails/GORM中可以找到DSL的另一个很好的例子。比如，使用下面的Groovy语法，可以在一个对象的属性上指定数据约束条件：

```
class State
{
        String twoLetterCode
        static constraints = {
                twoLetterCode unique: true, blank: false, size: 2..2
        }
}
```

对于表达约束的这种流畅而且表现力很好的语法，Grails能够聪明地识别，进而为前端和后端生成验证逻辑。

Groovy生成器（参见第17章）是DSL的很好的例子，它们非常流畅，而且是基于上下文构建的。

## 19.3　DSL 的分类



在设计一门DSL时，必须确定要设计的是哪种类型的DSL：外部的还是内部的。

外部DSL定义了一门新语言。我们可以灵活地选择语法，之后是解析新语言中的命令并执行动作。当我刚开始做我的第一份工作时，公司让我维护一门DSL，它需要大量使用`lex`和`yacc`。

（起初我还以为，之所以让我做，是因为我比较优秀。后来才明白，只是没有人愿意做而已。）解析真是充满“乐趣”！可以使用诸如C++和Java等语言，使用它们提供的解析功能及库，以及利用它们提供的大量解析功能和库的支持来处理繁重的任务。比如可以使用ANTLR来构建DSL（参见Terence Parr的*The Definitive ANTLR Reference: Building Domain-Specific Languages*[Par07]）。

内部的DSL，也称作嵌入式DSL，同样定义了一门语言，但是语法受到现有语言的约束。不必使用任何解析器，但必须巧妙地将语法映射到底层语言中的方法和属性等构造。内部DSL的用户可能意识不到自己正在使用的是一门更广的语言的语法。然而，为了让底层的语言为我们工作，创建内部的DSL需要在设计上付出巨大的努力，而且还需要一些聪明的技巧。

前面提到过Ant和Gant：前者使用的是XML，这是外部DSL的一个例子；而后者则使用Groovy来解决同样的问题，但它是内部DSL的一个例子。

## 19.4　设计内部的 DSL

动态语言很适合设计和实现内部的DSL。这些语言提供了很好的元编程能力和灵活的语法，便于轻松地加载和执行代码片段。

然而并非所有的动态语言都是一样的。

比如，我发现在Ruby中创建DSL非常容易。Ruby是动态类型的，括号可选，可以用冒号代替双引号引用字符串，等等。Ruby的优雅为创建内部DSL提供了极大的支持。

在Python中创建DSL就面临一些挑战。空白在Python中是有意义的，这可能会成为创建DSL的障碍。

Groovy的动态类型和元编程能力对于创建内部的DSL帮助很大。然而，如果挑剔的话，Groovy对括号的处理，以及它没有Ruby提供的优雅的冒号符号，都是其不足。对于这些限制，必须采取一些变通措施，后面将会介绍。

设计一门内部DSL，需要耗费很多时间，也需要付出极多的耐心与精力。要想成功，一定要有创造力，巧妙地处理问题，并且愿意做出妥协。

## 19.5　Groovy 与 DSL

Groovy的很多核心功能对创建内部DSL很有帮助，包括：

- 动态类型与可选类型（参见3.5节）；
- 动态加载、操纵和执行脚本的灵活性（参见10.8节）；
- 因为分类和`ExpandoMetaClass`，可以打开类（参见第13章）；
- 闭包为执行提供了很好的上下文（参见第4章）；
- 操作符重载有助于自由地定义操作符（参见2.8节）；

- 生成器支持（参见第17章）；
- 灵活的括号。

Groovy对括号的灵活处理，让人喜忧参半。调用需要参数的方法时，Groovy不要求括号；但如果调用的方法没有参数，则括号不可或缺。19.9节介绍了一个简单的技巧，用于解决这一烦恼。

本章接下来将介绍一个在Groovy中使用这些功能创建DSL的例子。

## 19.6 使用命令链接特性改进流畅性

利用Groovy支持将命令或方法调用链接起来的特性，可以实现一定程度的流畅性。当调用的方法需要参数时，Groovy不要求使用括号。此外，如果方法会返回一个结果，不使用点符号（.），就能在返回的这个实例上进行连续的调用。使用简单的、最普通的Groovy代码，不需要元编程的魔力，就可以像下面这样执行流畅的代码：

**CreatingDSLs/CommandChain.groovy**

```
move forward and then turn left
jump fast, forward and then turn right
```

看上去像个数据文件，但这是百分之百可以执行的Groovy代码。下面分析这段代码，并确定它执行所需的其余Groovy代码。

第一行没有一个逗号。Groovy将读取`move`和`forward`，并假定我们是在调用一个`move()`方法，`forward`是一个参数。我们定义一个`move()`方法，同时提供一个名为`forward`的变量，其中保存着我们期望的值，如forward。在Groovy处理完前两个单词后，它期待有一个可以调用`and()`方法的对象。为促成这一点，可以从`move()`方法返回一个支持`and()`方法的对象。因为第二行使用了一个逗号，`jump()`方法将接收两个参数。可以继续分析，以确定需要的方法、变量和参数。要处理前面像数据一样的代码，需要像下面这样创建一堆变量和方法：

**CreatingDSLs/CommandChain.groovy**

```
def (forward, left, then, fast, right) =
  ['forward', 'left', '', 'fast', 'right']

def move(dir) {
  println "moving $dir"
  this
}

def and(then) { this }

def turn(dir) {
  println "turning $dir"
  this
}
```

```
def jump(speed, dir) {
  println "jumping $speed and $dir"
  this
}
```

定义所需变量时使用了多赋值。move()、and()、turn()和jump()均返回this，也就是调用这些方法的对象。这使将方法在这两行中漂亮地链接起来成为可能。

将前面的代码与那两行流畅的文字放到一个文件中，使用groovy命令执行，输出如下：

```
moving forward
turning left
jumping fast and forward
turning right
```

Groovy中的命令链接特性，使得创建相当流畅的简单DSL非常容易。要创建更复杂的DSL，并在一个上下文内执行它们，还需要其他能力，下一节将予以介绍。

## 19.7　闭包与 DSL

with()方法可以在一个闭包内辅助实现委托调用，并提供一个执行上下文。可以利用这种方式创建自己的方法，兼具上下文和流畅性。

再来看一下订购比萨的例子。假如想创建一种自然流动的语法，但是不想创建一个PizzaShop实例，因为这样就太偏于实现细节了。我们希望上下文是隐式的。看看下面的代码（下一节将介绍如何让这段代码更流畅，更符合上下文驱动风格）：

**CreatingDSLs/ClosureHelp.groovy**

```
time = getPizza {
  setSize Size.LARGE
  setCrust Crust.THIN
  setTopping "Olives", "Onions", "Bell Pepper"
  setAddress "101 Main St., ..."
  setCard(CardType.VISA, "1234-1234-1234-1234")
}

printf "Pizza will arrive in %d minutes\n", time
```

getPizza()方法接受一个闭包，闭包内使用了PizzaShop类的实例方法来实现订购比萨功能。不过这里的PizzaShop实例是隐式的。delegate（参见4.9节）负责将方法路由到隐式的实例，这一点在下面getPizza()方法的实现中可以看到：

**CreatingDSLs/ClosureHelp.groovy**

```
def getPizza(closure) {
  PizzaShop pizzaShop = new PizzaShop()
  closure.delegate = pizzaShop
```

```
  closure()
}
```

执行调用getPizza()的代码，输出如下：

```
Pizza will arrive in 25 minutes
```

稍等一下，输出中打印的time的值是怎么得到的？因为getPizza()中的最后一条语句是调用闭包，所以闭包返回什么，这个方法就返回什么。而闭包内的最后一条语句是setCard()，所以该方法的结果会被返回给调用者。这里DSL强加了顺序信息：在订购比萨时，setCard()必须是最后调用的方法。可以致力于改进接口，让订购更为显而易见。此外，也可以将像setSize Size.LARGE这样的设置语句修改为像size = Size.LARGE这样的赋值语句。

## 19.8　方法拦截与 DSL

不使用PizzaShop类，也可以实现订购比萨的DSL，比如可以完全靠拦截方法调用来实现。下面从订购比萨的代码入手（保存在一个名为orderPizza.dsl的文件中）：

**CreatingDSLs/orderPizza.dsl**

```
size large
crust thin
topping Olives, Onions, Bell_Pepper
address "101 Main St., ..."
card visa, '1234-1234-1234-1234'
```

这怎么看都不像代码，倒是更像一个数据文件。然而，这就是纯正的Groovy代码，而且我们正打算执行它（文件中可见的一切，除了双引号中的字符串，其他不是方法名就是变量名）。但是在此之前，还必须玩些技巧，也就是设计自己的DSL。

首先创建一个名为GroovyPizzaDSL.groovy的文件，在其中定义large、thin和visa等变量（可以随意定义其他变量，比如small、thick和masterCard等）。现在定义一个acceptOrder()方法，用于调用最终执行DSL的闭包。此外，实现methodMissing()方法，对于不存在的方法，该方法会被调用（在DSL文件orderPizza.dsl中调用的方法几乎都不存在）。

**CreatingDSLs/GroovyPizzaDSL.groovy**

```
def large = 'large'
def thin = 'thin'
def visa = 'Visa'
def Olives = 'Olives'
def Onions = 'Onions'
def Bell_Pepper = 'Bell Pepper'

orderInfo = [:]
def methodMissing(String name, args) {
  orderInfo[name] = args
```

```
}

def acceptOrder(closure) {
  closure.delegate = this
  closure()
  println "Validation and processing performed here for order received:"
  orderInfo.each { key, value ->
    println "${key} -> ${value.join(', ')}"
  }
}
```

必须想办法把这两个脚本放到一起执行。这可以非常简单地实现（参见10.8节），如下所示。调用`GroovyShell`，加载前面的两个脚本，将它们聚合到一起，形成一个脚本，然后计算处理。

**CreatingDSLs/GroovyPizzaOrderProcess.groovy**

```
def dslDef = new File('GroovyPizzaDSL.groovy').text
def dsl = new File('orderPizza.dsl').text

def script = """
${dslDef}
acceptOrder {
  ${dsl}
}
"""

new GroovyShell().evaluate(script)
```

前面代码的输出如下：

```
Validation and processing performed here for order received:
size -> large
crust -> thin
topping -> Olives, Onions, Bell Pepper
address -> 101 Main St., ...
card -> Visa, 1234-1234-1234-1234
```

可见，如果知道如何利用Groovy的MOP能力，在Groovy中设计与执行DSL相当容易（就像我们在`orderpizza.dsl`中所做的那样）。

## 19.9　括号的限制与变通方案

先放下比萨的例子，来看一个简单的计数器。计数器是一个支持计算总数的设备，本节将演示如何为一个简单的计数器创建一门DSL。下面是第一次尝试：

**CreatingDSLs/Total.groovy**

```
value = 0
def clear() { value = 0 }
def add(number) { value += number }
```

```
def total() { println "Total is $value" }

clear()
add 2
add 5
add 7
total()
```

前面代码的输出如下：

```
Total is 14
```

这段代码中写的是`total()`和`clear()`，而不是`total`和`clear`。下面去掉括号，尝试调用`total`：

**CreatingDSLs/Total.groovy**

```
try {
  total
} catch(Exception ex) {
  println ex
}
```

执行这段代码，得到如下结果：

```
groovy.lang.MissingPropertyException:
  No such property: total for class: Total
```

Groovy认为对`total`的调用引用了一个（不存在的）属性。使用一门语言来设计DSL就像陪两岁的孩子玩耍，当孩子发脾气时，不要和他争，得让着他点。因此，在这种情况下，告诉Groovy一切正常，然后把问题处理掉。简单地创建它想要的属性即可：

```
value = 0
def getClear() { value = 0 }
def add(number) { value += number }
def getTotal() { println "Total is $value" }
```

通过编写`getTotal()`和`getClear()`方法，实现了名为`total`和`clear`的属性。现在Groovy会非常高兴（像个孩子一样）地跟我们玩了，我们也可以不用括号调用这些属性了：

```
clear
add 2
add 5
add 7
total
clear
total
```

输出如下：

```
Total is 14
Total is 0
```

前面介绍了创建流畅的语法的不同方式，下面将探讨如何拦截和合成DSL中的方法调用。

## 19.10　分类与 DSL

使用分类，可以以可控的方式拦截方法调用（参见13.1节）。在创建DSL时也可以使用分类。现在想办法实现下面这种流畅的调用：`2.days.ago.at(4.30)`。

2是一个`Integer`实例，而`days`不是这个实例上的属性。这里就使用分类将其注入为一个属性（对应`getDays()`方法）。`days`在这里看就是噪音，但是在需要区分五天以前还是五分钟以前的另一个上下文中，可能就是有用的了。在two days ago at 4.30这个句子中，它起到的是连接作用。可以将`getDays()`方法实现为接受`Integer`并返回接受的对象。`getAgo()`方法对应`ago`属性，它接受一个`Integer`实例，使用`Calendar`上的操作，根据当前的日期计算`Integer`实例所指的天数之前的日期，然后将这个日期返回。最后，`at()`方法将该日期上的时间设置为参数（`4.30`）指定的时间，然后返回一个`Date`实例。这一切可以都在`use()`块内执行，如下面代码所示。（这里没有在作为参数提供的时间上执行错误检查，如果喜欢，可以发送4.70来代替5:10；这是一个没有写入文档的特性。此外，我们可能希望复制调用`at()`方法的`Calendar`实例，以避免任何副作用。）

**CreatingDSLs/DSLUsingCategory.groovy**

```
class DateUtil {
  static int getDays(Integer self) { self }

  static Calendar getAgo(Integer self) {
    def date = Calendar.instance
    date.add(Calendar.DAY_OF_MONTH, -self)
    date
  }

  static Date at(Calendar self, Double time) {
    def hour = (int)(time.doubleValue())
    def minute = (int)(Math.round((time.doubleValue() - hour) * 100))
    self.set(Calendar.HOUR_OF_DAY, hour)
    self.set(Calendar.MINUTE, minute)
    self.set(Calendar.SECOND, 0)
    self.time
  }
}

use(DateUtil) {
  println 2.days.ago.at(4.30)
}
```

前面代码的输出如下：

```
Thu Jan 31 04:30:00 MST 2008
```

这里创建的DSL语法还有最后一个问题：用的是`2.days.ago.at(4.30)`。使用`4:30`来代替

4.30会更自然，因此最好是修改为使用2.days.ago.at(4:30)。Groovy可以接受Map作为方法的一个参数。

通过将at()方法的参数定义为Map，而非Double，可以实现上面的需求：

**CreatingDSLs/DSLUsingCategory2.groovy**

```
class DateUtil {
  static int getDays(Integer self) { self }

  static Calendar getAgo(Integer self) {
    def date = Calendar.instance
    date.add(Calendar.DAY_OF_MONTH, -self)
    date
  }

  static Date at(Calendar self, Map time) {
    def hour = 0
    def minute = 0
    time.each {key, value -> hour = key.toInteger()
      minute = value.toInteger()
    }
    self.set(Calendar.HOUR_OF_DAY, hour)
    self.set(Calendar.MINUTE, minute)
    self.set(Calendar.SECOND, 0)
    self.time
  }
}

use(DateUtil) {
  println 2.days.ago.at(4:30)
}
```

这段代码的输出如下：

```
Thu Jan 31 04:30:00 MST 2008
```

分类方式唯一的限制是，只能在use()块内使用该DSL。这一限制可能实际上也是优势，因为方法注入是可控的。一旦离开代码块，注入的方法就会被从上下文中丢弃，不再可用，这可能比较理想。下一节将介绍如何使用ExpandoMetaClass实现同样的语法。

## 19.11 ExpandoMetaClass 与 DSL



分类只能应用于use块内，而且其效果被限制在了作用域内。如果希望方法注入在整个应用内都有效果，可以使用ExpandoMetaClass来代替分类。下面使用ExpandoMetaClass实现上一节介绍的DSL语法：

**CreatingDSLs/DSLUsingExpandoMetaClass.groovy**

```
Integer.metaClass{
  getDays = { ->
    delegate
  }

  getAgo = { ->
      def date = Calendar.instance
      date.add(Calendar.DAY_OF_MONTH, -delegate)
      date
    }
}

Calendar.metaClass.at = { Map time ->
    def hour = 0
    def minute = 0
    time.each {key, value ->
      hour = key.toInteger()
      minute = value.toInteger()
    }

    delegate.set(Calendar.HOUR_OF_DAY, hour)
    delegate.set(Calendar.MINUTE, minute)
    delegate.set(Calendar.SECOND, 0)
    delegate.time
}

println 2.days.ago.at(4:30)
```

这里将想要的方法添加到了`Integer`类和`Calendar`类的`ExpandoMetaClass`中。调用这些流畅的方法，会被路由到我们添加的方法，如下所示：

```
Fri Feb 03 04:30:00 MST 2012
```

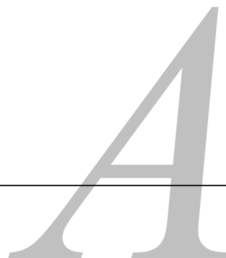
使用`ExpandoMetaClass`添加方法这种解决方案，要比编写分类所用的静态方法清晰得多。

现在知道了在Groovy中创建内部的DSL相当容易。动态特性和可选类型对于创建流畅的接口帮助很大。闭包可以帮助创建上下文。Groovy的分类和`ExpandoMetaClass`对于方法调用与属性的注入、拦截和合成也很有帮助。最后，Groovy能够加载和执行任意脚本，这在执行DSL时可以派上用场。

希望本书中对Groovy这一强大的动态语言及其功能的介绍会使您的阅读和学习犹如一段颇为享受的美好旅程。我自己已经将Groovy应用到一切任务中，小到执行自动化例行任务的小型脚本，大到运行企业级应用。Groovy的简洁以及支持与Java轻松集成吸引了我，而由此带来的开发效率的提高留住了我。由衷地希望本书中的概念能够帮助你使用Java虚拟机上的这门强大到不可思议的语言编写可靠的程序，同时获得动态语言的开发效率。祝你一切顺利！

附录 A

# Web资源



A Bit of Groovy History：http://glaforge.free.fr/weblog/index.php?itemid=99，Guillaume Laforge谈Groovy历史的一篇博客文章。

API for `FactoryBuilderSupport`： http://groovy.codehaus.org/api/groovy/util/FactoryBuilderSupport.html，`FactoryBuilderSupport`类的API，它是`SwingBuilder`的新基类。

ASTTest Annotation： http://groovy.codehaus.org/gapi/groovy/transform/ASTTest.html，用于测试和调试AST变换的Groovy注解。

CodeNarc：http://codenarc.sourceforge.net，CodeNarc是一款基于Groovy的静态代码分析工具。

Crash of the Mars Orbiter：http://www.cnn.com/TECH/space/9909/30/mars.metric.02，CNN有关火星轨道探测器坠毁的报道。

Duck Typing.： http://c2.com/cgi/wiki?DuckTyping，什么是鸭子类型。

easyb： http://www.easyb.org easyb，是一款自动测试工具，支持流畅地进行功能测试和继承测试。

Eclipse Plug-in for Groovy：http://groovy.codehaus.org/Eclipse+Plugin Eclipse，IDE中用于支持Groovy开发的插件。

`FactoryBuilderSupport` ： http://groovy.codehaus.org/FactoryBuilderSupport ， Groovy 的`FactoryBuilderSupport`类，它是`SwingBuilder`的新基类。

Gant Home：http://gant.codehaus.org，Gant的网站，它是一款类似Ant的工具，不过它使用的是Groovy，而非XML。

The GDK.：http://groovy.codehaus.org/groovy-jdk，列出了Groovy JDK中的方法。

Getting Started with Grails： http://www.infoq.com/minibooks/grails，Jason Rudolph介绍Grails使用的一本书。

Good, Bad, and Ugly of Java Generics：http://www.agiledeveloper.com/articles/GenericsInJava

PartI.pdf，探讨Java泛型的优点、缺点以及丑陋之处的一篇文章。

GPars：http://gpars.codehaus.org，GPars库提供了很多并发编程选择。

Gradle：http://gradle.org，Gradle，是一款轻量级的构建管理工具，可以帮助程序员轻松配置构建，只需要很少的配置信息，不需要XML。

Grails Home： http://grails.org/，Grails项目主页，提供了文档及项目下载。

Griffon Project： http://griffon.codehaus.org，用于构建桌面应用的Groovy框架。

Groovy API Javadoc： http://groovy.codehaus.org/api，Groovy API的Javadoc帮助文件。

Groovy Closures Definition：http://groovy.codehaus.org/Closures++Formal+Definition，Groovy闭包的探讨与定义。

GroovyCollectionsSupport：http://groovy.codehaus.org/groovy-jdk/java/util/Collection.html，Groovy向集合类添加的扩展与特性。

Groovy Daily Build：http://build.canoo.com/groovy，Groovy项目的当前构建版本，紧跟Groovy最新趋势的程序员可以下载。

Groovy Download Page：http://groovy.codehaus.org/Download，Groovy最新发布版本和之前版本的下载链接。

Groovy Home： http://groovy.codehaus.org，Groovy项目主页，提供了文档和下载。

Groovy Looping： http://groovy.codehaus.org/Looping，介绍了Groovy中实现循环的不同方式。

Groovy Mailing Lists： http://groovy.codehaus.org/Mailing+Lists，Groovy邮件列表。

Groovy Operator Overloading：http://groovy.codehaus.org/Operator+Overloading，Groovy的操作符重载以及操作符所映射的方法。

Groovy Scriptom API： http://groovy.codehaus.org/COM+Scripting，支持与 Windows ActiveX和COM交互的Groovy API。

Groovy String Support：http://groovy.codehaus.org/groovy-jdk/java/lang/String.html，Groovy中对`String`的扩展与支持。

Groovy's Support for java.math Classes： http://groovy.codehaus.org/Groovy+Math，Groovy为提供更高的精度对java.math的支持。

Groovy's Support for Map： http://groovy.codehaus.org/groovy-jdk/java/util/Map.html，Groovy向Java的`Map`添加的扩展和特性。

GVM：http://gvmtool.net，用于管理多个Groovy版本和相关工具的一款工具。

Higher-Order Function：http://c2.com/cgi/wiki?HigherOrderFunction，高阶函数的相关讨论。

IntelliJ IDEA：http://www.jetbrains.com/idea，一款流行的Java IDE，对Groovy提供了出色的支持。

Java Download：http://www.oracle.com/technetwork/java/javase/downloads/index.html，Java和JDK的下载页面。

JRuby Home：http://jruby.codehaus.org，JRuby项目主页，提供了文档和下载。

Languages and Idioms：http://blog.agiledeveloper.com/2007/05/its-not-languages-but-their-idioms-that.html，探讨语言与习惯用法的一篇博客文章。

Markmail for Groovy Mailing List：http://groovy.markmail.org，这里可以方便地搜索Groovy用户邮件列表中曾经讨论过的主题。

MetaClass and Method Interception：http://graemerocher.blogspot.com/2007/06/dynamic-groovy-groovys-equivalent-to.html，Graeme Rocher介绍Groovy的元编程能力以及打开类的一篇博客文章。

Mocks Aren't Stubs：http://martinfowler.com/articles/mocksArentStubs.html，Martin Fowler，谈模拟与存根的相似性及差别。

No Fluff Just Stuff.：http://www.nofluffjuststuff.com，经常在不同地点举行的一个很受欢迎的Java会议。

The Official Website for the Book：http://pragprog.com/titles/vslg2，本书的官方网站，可以下载本书的示例源代码，查阅勘误信息以及提供反馈。

The Pragmatic Programmers：http://pragprog.com，本书英文版出版商的网站。

Runtime vs. Compile Time/Static vs. Dynamic：http://groovy.codehaus.org/Runtime+vs+Compile+time+Static+vs+Dynamic，有关Groovy对动态类型的支持的探讨和理论介绍。

Selenium：http://seleniumhq.org/download，可以测试Web应用的自动化工具。

Some Differences Between Java and Groovy：http://groovy.codehaus.org/ Differences+from+Java，详细介绍了Java和Groovy的一些不同。

Spock Library：http://spockframework.org，Spock测试库的主页。

State of IDE Support for Groovy：http://groovy.codehaus.org/IDE+Support，支持Groovy开发的不同IDE，以及它们目前的支持程度。

Sun/Java Scripting Project Home：https://scripting.dev.java.net，有关脚本语言和JSR 223的详细介绍。

Technical Debt：http://martinfowler.com/bliki/TechnicalDebt.html，Martin Fowler谈技术债。

TextMate：http://macromates.com，Mac上一款流行的编辑器TextMate。

TextMate Groovy Bundle：http://docs.codehaus.org/display/GROOVY/TextMate，支持TextMate的Groovy bundle。

Treating a Java Method as a Closure：http://www.jroller.com/melix/entry/coding_a_groovy_closure_in，Champeau介绍了如何在Groovy端将一个Java方法看作一个闭包。

Tweaking the Groovy Bundle for TextMate Editor：http://tinyurl.com/ywotsj，Venkat的一篇博客文章，介绍了如何修改Groovy bundle，使输出更为方便快捷。

Using JUnit 4 with Groovy：http://groovy.codehaus.org/Using+JUnit+4+with+Groovy，在Groovy中使用JUnit 4.0的步骤。

Using Notepad2：http://tinyurl.com/yqfucf，介绍在Windows上如何使用Notepad2编辑和运行Groovy的一篇博客文章。

Why Copying an Object Is a Terrible Thing to Do：http://www.agiledeveloper.com/articles/cloning072002.htm，探讨Java中的对象复制问题的一篇文章。

Why Getter and Setter Methods Are Evil：http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html，Allen Holub的一篇文章。

Why Scripting Languages Matter：http://www.oreillynet.com/pub/wlg/3190，Tim O'Reilly探讨了应用的特性及脚本语言所起的作用。

Xerces XML Parser：http://xerces.apache.org/xerces-j，基于Java的一款流行的XML解析器。

附录 B

# 参考书目

B

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2nd, 1996.（《计算机程序的构造和解释》（原书第2版），机械工业出版社。）

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.（《测试驱动开发：实战与模式解析》，机械工业出版社。）

[Bec96] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, Reading, MA, 2008.（《Effective Java中文版（第2版）》，机械工业出版社。）

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman, Reading, MA, 1990.

[Eck06] Bruce Eckel. *Thinking in Java*. Prentice Hall, Englewood Cliffs, NJ, Fourth, 2006.（《Java编程思想（第4版）》，机械工业出版社。）

[Fri97] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, 1997.（《精通正则表达式（第3版）》，电子工业出版社。）

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.（《设计模式：可复用面向对象软件的基础》，电子工业出版社。）

[Gra07] James Edward Gray II. *TextMate: Power Editing for the Mac*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2007.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley Longman, Reading, MA, Third, 1997.（《计算机程序设计艺术 第1卷 基本算法（第3版）》，人民邮电出版社。）

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning

Publications Co., Greenwich, CT, 2003.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, Second, 1997.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2007.

[Rai04] J. B. Rainsberger. *JUnit Recipes: Practical Methods for Programmer Testing*. Manning Publications Co., Greenwich, CT, 2004.（《Junit Recipes中文版——程序员实用测试技巧》，电子工业出版社。）

[Seb04] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, Reading, MA, 2004.（《编程语言原理（第10版）》，清华大学出版社。）

[TH03] David Thomas and Andrew Hunt. *Pragmatic Unit Testing In Java with JUnit*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2003.（《单元测试之道Java版——使用Junit》，电子工业出版社。）

# 看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks